Summarizing Time-Evolving Data

by

Inés Fernando Vega López

A Dissertation Submitted to the Faculty of the

Department of Computer Science

In Partial Fulfillment of the Requirements
For the Degree of

Doctor of Philosophy

In the Graduate College

The University of Arizona

2004

THE UNIVERSITY OF ARIZONA ®
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Ines Fernando Vega-Lopez entitled Summarizing Time-Evolving Data

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

Bongki Moon	May 13, 2004 Date
Kobus Barnard	May 10, 2004 Date
Richard Snodgrass	May 10, 2004 Date
Kurt Fenstermacher	5/10/2004 Date
	Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

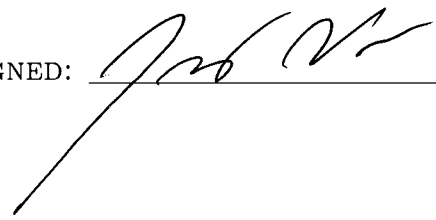
I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director Bongki Moon	May 13, 2004 Date

Statement by Author

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Signed: 

To my wife, Amelia, and my parents Rito and Martha. For all your love, encouragement and support.

To Sofia and Fernando. For letting me spend the time I should have been with you achieving my dreams. I will make it up to you guys.

Acknowledgments

I need to express my special thanks to my advisor, Doctor Bongki Moon, for his unconditional support and excellent guidance during my doctoral studies at the University of Arizona. It has been a great learning experience and a privilege to work with him for the past years.

I also need to thank my dear friends at the department of Computer Science, Praveen, Quanzhong, Mohan, Somu, Sriraman, Cesim, Joe, and Arvind, for making graduate school a nice place to be. You always had the finest attentions to me and to my family and I really appreciate it.

Finally, my studies at the University of Arizona would have not been possible without the financial support of the Mexican Foundation for Science and Technology (CONACyT, Grant No. 117476), the Autonomous University of Sinaloa, and the National Science Foundation (Grants No. IIS-0100436 and EIA-0080123).

Table of Contents

TABLE OF CONTENTS—*Continued*

TABLE OF CONTENTS—*Continued*

List of Figures

List of Tables

Abstract

We live in a highly dynamic environment where we have learned to appreciate time as a very important aspect in our lives. The world, as we know it, is not the same as it was yesterday and it will not be the same tomorrow. Things evolve over time. As we try to better understand our environment, we need to model this continuous change. To satisfy this need, the research community has developed temporal database systems.

Summarizing time-evolving data is a challenging problem. Not only is this problem challenging because of the large size of the data sets usually involved but also because the temporal ordering and validity of every entry in the database must be considered. In this dissertation we present effective and efficient solutions to the problem of summarizing time-evolving data from two different perspectives. The first perspective considers capturing the collective behavior of temporal data in a process known as *temporal aggregation*. For this, we propose a model that reduces the evaluation of temporal aggregation queries to the problem of selecting qualifying tuples and grouping these tuples into collections to which an aggregate function is to be applied. In addition, we propose IO efficient algorithms for the evaluation of temporal aggregation queries. The second perspective we study is aimed toward the detection of individual entities in the database whose temporal behavior (evolution) matches a given pattern. The process of matching the temporal evolution of an entry in the database to a query pattern is known as *similarity search*. To address this problem, we propose a new indexing paradigm called *Skyline Index* and a new and compact representation of time series called *Self COntained Bit Encoding (SCoBE)*.

The techniques proposed in this dissertation provide significant performance improvements over the current state-of-the-art. Our empirical evidence shows that the proposed algorithms for the evaluation of temporal aggregation queries significantly

outperform previous approaches. We experimentally show that the Skyline Index can be coupled with the state of the art dimensionality reduction techniques and significantly improve the performance on the evaluation of similarity search queries. Similarly, we show that *SCoBE* consistently outperforms previously proposed transformations of time series data for the evaluation of similarity search queries under a variety of scenarios.

Chapter 1
Introduction

We live in a highly dynamic environment where we have learned to appreciate time as a very important aspect in our lives. The world, as we know it, is not the same as it was yesterday and it will not be the same tomorrow. Things evolve over time. As we try to better understand our environment, we need to model this continuous change.

Improvements in the storage and processing power of computer systems have allowed us to consider the time dimension as a critical component of a wide variety of scientific and business applications. The importance of modeling time has been recognized by the database research community and database models and query languages have been developed [66, 111, 116, 121]. Temporal applications are becoming more common with the ever-increasing capabilities of computer systems to store and process a large amount of data. Examples of such applications abound in a variety of fields such as business, medicine, physics, land management, weather monitoring, natural resources management, environmental, ecological, and biodiversity studies, tracking of mobile devices, and navigation systems.

A main characteristic of temporal applications is that the amount of data they store and process grows rapidly over time. For example, remotely sensed data from NASA are captured at a rate of several Gigabytes a day. Due to the size of these data sets, the study of individual entries in the database is rarely feasible. To make these huge collections of detailed data available to human analysts, there is a clear need for extracting general characterizations of subsets of the data [106]. Therefore, the development of techniques that efficiently summarize and discover trends in time-evolving data and help in decision making is of critical importance. To put this

problem in perspective, consider NASA's mission control. During a flight of a space shuttle, approximately 20,000 sensors are monitored once per second [73]. At this rate of sampling it is highly unlikely that a human operator can analyze all the received data. In the best case, only a small fraction of data can be viewed in real time. Instead of presenting large quantities of unprocessed data to human operators, it is more useful to only indicate when unexpected trends are registered by the on-board sensors. Raising flags about unexpected readings can help human operators in the process of decision making.

1.1 Motivation

Summarizing temporal data is a challenging problem. Not only is this problem challenging because of the large size of the data sets involved but also because the temporal ordering and validity of every entry in the database must be considered. In this dissertation, the problem of summarizing time-evolving data is studied from two perspectives. The first perspective considers capturing the information contained in a collection of temporal data in a single time-evolving value using a process known as *temporal aggregation.* The second perspective considers temporal data as sequences of changing values and it is aimed toward the detection of sequences in the database whose temporal behavior (evolution) matches a given pattern. The process of matching the temporal evolution of an entry in the database to a query pattern is known as *temporal pattern similarity search*, referred to hereafter as just similarity search.

Neither of these two approaches for summarizing time-evolving data can be trivially addressed. Yet, developing efficient techniques for addressing the problems derived from either of these perspectives has significant impact on several fields of knowledge. Consider a cardiologist studying electrocardiograms from several patients. For this particular application, it would be extremely useful to find strong correlations among the electrocardiograms. Based on such correlations, an specialist can timely

diagnose life-threatening diseases and provide appropriate treatment. Similarly, a vulcanologist studying the behavior of volcanoes nearby densely populated areas (*e.g.*, the Popocatepetl volcano, near México city) could issue early warnings of possible eruptions based on the evolution of the values recorded by the array of sensors monitoring the volcano. On the other hand, perhaps an analyst is not interested in the evolution of an individual but rather in the collective behavior of all the entities modeled in a database. Consider now a network administrator facing the decision of whether or not to buy additional bandwidth. In this case, the interest is in knowing what the maximum workload is, as well as its temporal behavior, rather than knowing what the bandwidth needs for a particular workstation are.

The need for understanding time evolving-data motivates the research presented in this dissertation. We study temporal aggregation when the need is to know the collective temporal behavior of data. We study similarity search for identifying individuals following a particular temporal behavior.

1.2 Temporal Aggregation

An aggregate function takes a set of tuples and returns a single value that summarizes the information contained in the set of tuples [53, 80, 134]. Aggregation is is the effect of applying an aggregate function to a set of qualifying tuples. Queries using aggregate functions are called aggregate queries. In applications such as data warehouses, on line analytical processing, and decision support systems, aggregate queries are used as a standard means of reducing the volume of the data [32]. This data reduction allows human operators to better understand the state of the entities modeled in the database and to make decisions accordingly. For instance, a store manager does not need to look at every sales transaction to evaluate the performance of his staff. Knowing the total amount of sales could be sufficient in this case. Aggregate queries are generally expensive to evaluate because they must account for every qualifying

entity (potentially the entire data set) in the relation. Thus efficient algorithms for their evaluation are required. Traditional (*i.e.*, non-temporal) aggregate queries are logically processed in two steps. First, data is collected from a database as specified by the selection predicates of the query. Then, the qualifying entities are grouped and aggregate functions are applied to each group [32].

The complexity of an aggregate query increases considerably when the time dimension is included. In temporal aggregation we represent the information contained in a database with a single value that changes over time. That is, the aggregate value changes as the collective information in the database evolves. To keep track of the evolution of the aggregate value on a temporal database we need to identify the time intervals for which the state of the database remains constant. Because there are no data changes during these intervals, they are called *constant intervals.* Once constant intervals have been identified, tuples valid at a particular constant interval form a group on which an aggregate function applied. The aggregate value obtained from this group of tuples is valid for the entire constant interval.

1.3 Similarity Search

The projection of the time-varying attributes of a temporal object onto the time dimension generates a time series representing the temporal evolution of such object. Consider the evolution of stock values, for example. A time series is, therefore, a potentially long sequence of values. Each of these values represents the measurement of an attribute of interest at a point in time.

Similarity search on time series is used when applications need to compare the object's temporal evolution to a particular pattern known as the query pattern. For example, a fund manager of a stock brokerage firm may be interested in finding all stocks whose prices moved similarly to that of a particular stock or following a certain pattern (*e.g.*, head-and-shoulder). The main idea behind temporal pattern similarity

search is to identify those objects in the database whose evolution matches the query pattern [28, 39, 65]. This search involves some criterion that quantifies the degree of similarity between two patterns [109]. The need for modeling similarity search has influenced the development of extensions to query languages for supporting complex matching queries [58]. At the same time, the need for efficiently processing similarity search queries has motivated a considerable amount of research. In particular, the problem of indexing and searching time series data has been the focus of many research activities in the database community for the past few years. An outstanding example is the seminal work of Agrawal *et al.* [4].

1.4 Contributions of this Dissertation

This dissertation addresses the problem of how to efficiently summarize time-evolving data. This problem is studied from two different perspectives where the focus is either on finding the collective behavior observed in the database or on finding individuals whose temporal behavior matches a particular pattern. Regarding the need for efficient mechanisms to capture the collective behavior of time-evolving data (*i.e.*, temporal aggregation), the contributions of this dissertation are :

- A new model for studying aggregation queries is defined. This model allows the classification of previous research work, on one hand, and the detection of research opportunities, on the other.

- New IO-efficient algorithms for the evaluation of temporal aggregation queries are introduced. These algorithms can be easily parallelized with the help of a data structure called the *meta array*. The meta array allows the proposed parallel algorithms to operate with minimal data replication.

Regarding the search for individuals in the database matching a pattern of interest (*i.e.*, similarity search), the contributions of this dissertation are:

- A new paradigm for the grouping of time series in a hierarchical index is introduced. This paradigm, called *Skyline Bounding Regions*, represents a group of time series based on their collective shape. Empirical evaluation presented in this dissertation shows that this new paradigm achieves considerable performance improvements over previous approaches.

- A new representation for evaluating similarity search queries on time series data is introduced. This new data representation, based on quantization, is called *Self COntained Bit Encoding* (*SCoBE*). Experimental evidence shows that *SCoBE* significantly outperforms previously proposed techniques for the evaluation of similarity search queries. These performance improvements are observed for whole sequence match, subsequence match, and queries using dynamic time warping.

1.5 Organization of this Dissertation

The rest of this dissertation is organized as follows. Chapter 2 introduces a model for describing aggregation queries. Previous research work on aggregation is also presented at that chapter. We present new algorithms for the efficient evaluation of temporal aggregate queries in Chapter 3. Chapter 4 provides a description of previous work addressing similarity search queries on time series data. We present the *Skyline index*, an index-based approach for the evaluation of whole sequence match queries, in Chapter 5, whereas Chapter 6 introduces *SCoBE*, a time series representation that allows efficient evaluation of sequence match queries. Finally, Chapter 7 presents the conclusions of this dissertation, as well as visions for future work.

Chapter 2
Survey: Capturing the Collective Behavior of Data

In studying previous research on aggregation we have observed inconsistencies in the problem definition, as well as in the terms used to refer to specific concepts. Therefore, we start with a concrete definition of the problem of aggregation on databases and present a model to describe aggregation queries. Such model can be applied to traditional databases, in which objects lack temporal extent, as well as to temporal databases. If necessary, this model can be extended to handle spatial, and spatiotemporal objects. This allows us to classify previous work on aggregation and to identify areas of this problem that have not yet been addressed by the research community.

2.1 Aggregate Functions

Aggregate functions are widely used in database applications. Their popularity is reflected in the presence of aggregates in a large number of queries in the decision support benchmark *TPC-D* [52]. The ability of aggregate functions to provide summarized information from a large collection of data is indeed fundamental in specific, increasingly relevant, application domains such as data warehousing, On Line Analytical Processing (OLAP), decision support, statistical evaluation, and management of geographical data [18, 24, 47, 54, 59, 128].

An *aggregate function* takes a set of tuples and returns a single value that summarizes the information contained in the set of tuples [18, 53, 80]. In the context of this dissertation, *aggregation* is the effect of applying an aggregate function to a group of qualifying tuples. Aggregate functions have received several names in the

extent research literature. Epstein differentiates between *scalar aggregate* and *aggregate function* to distinguish between queries returning single or multiple aggregate values [41]. In Epstein's work, an aggregate query can return more than one value if the tuples are first partitioned into disjoint subsets based on a grouping attribute (*i.e.*, a `GROUP-BY` query). The SQL standard uses the term *set functions* to refer to aggregate functions. In the rest of this work, we refer to the functions computing an aggregate value from a set of tuples as aggregate functions. As we will see, there is no need to differentiate between functions returning either single or multiple aggregate values. Multiple aggregate values are the result of an orthogonal operator that partitions the relation into sets of tuples also known as aggregation groups. We favor the term *aggregate function* over *set function* used by SQL because it is more specific.

The SQL standard provides a variety of aggregate functions. The SQL-92 standard includes five such functions, namely `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX` [86]. The SQL:1999 standard adds `EVERY`, `SOME`, and `ANY`, whereas the SQL/OLAP addendum [1] to the SQL:1999 standard includes 18 additional aggregate functions [86].

2.1.1 Other Notions of Aggregation

Aggregation in databases has different meanings depending on the context. For instance, it could refer to aspects of language design, data modeling, or the evaluation of aggregate functions. For data modeling, aggregation is a form of abstraction in which a relationship between objects is considered as a higher level (aggregate) object [103, 104]. Our focus is on the study of algorithms for computing aggregate functions on temporal objects. We do not consider issues of language design or data modeling further.

[1] The aggregate functions defined in SQL/OLAP are `STDDEV_POP`, `STDDEV_SAMP`, `VAR_POP`, `VAR_SAMP`, `COVAR_POP`, `COVAR_SAMP`, `CORR`, `REGR_SLOPE`, `REGR_INTERCEPT`, `REGR_COUNT`, `REGR_R2`, `REGR_AVGX`, `REGR_AVGY`, `REGR_SSX`, `REGR_SSY`, `REGR_SYY`, `PERCENTILE_DISC`, and `PERCENTILE_CONT`.

Name	*DepartmentId*	*Age*	*Salary*
Praveen	1	24	45000
John	1	28	42000
Frank	1	50	80000
Sophia	2	25	40000
Fernando	2	30	48000

TABLE 2.1. Employees: A Sample Relation

2.2 Traditional Aggregation

Aggregate functions are applied to a collection (*i.e.*, set) of tuples. Klug [80] provided a formal framework for defining aggregate functions for relational databases. In his model, for a relation with n attributes, he proposed to use a set of n aggregate functions, each function defined on one attribute of the relation. Formally, for a relation R with schema $\{A_1, A_2, \ldots, A_n\}$, with each attribute A_i associated with domain D_i, a countable set $Agg = \{f_1, f_2, \ldots, f_n\}$ of aggregate functions should exist. Each function $f_i \in Agg$ operates on attribute $A_i \in R$, and for each function $f_i \in Agg$, $f_i : D_1 \times D_2 \times \ldots D_n \rightarrow D_{agg}$, where D_{agg} is the domain of the aggregate function. Here, we present a framework for analyzing aggregate queries based on the mechanisms used to define collections of tuples. Given a relation, a collection of tuples can be generated at three different levels. For example, consider the relation **employees** given in Table 2.1, whose schema is $\{Name, DepartmentId, Age, Salary\}$. A typical aggregate query on this relation could be the following.

Query 1. *Compute the highest salary on each department.*

The results for Query 1 are shown in Table 2.2. In this case, the `MAX` aggregation function is applied to collections of tuples created by a process known as *group composition.* In group composition, tuples sharing the same values in a list of attributes (termed *grouping attributes*) form a collection. Because these collections of tuples result from grouping composition, let us call them *groups.* Aggregate functions are then

DepartmentId	MAX(*Salary*)
1	80000
2	48000

TABLE 2.2. The Result of an Aggregate Query Using Group Composition and the MAX Aggregate Function

applied to each group of tuples. This procedure aggregates information at a coarse level because a single aggregate value is generated for each group. In the case of Query 1, two groups of tuples were generated by the distinct values of *DepartmentId* (*i.e.*, the grouping attribute). The first group is composed the the tuples corresponding to employees Praveen, John, and Frank from whom, Frank has the highest salary (80000). The second group is composed by the tuples corresponding to Fernando and Sofia. Fernando has the highest salary of the group (48000). Therefore, the result for this query is the relation formed by the tuples $\{< 1, 80000 >, < 2, 48000 >\}$.

Different queries may request to generate aggregate values at a finer level. Consider for example the following query.

Query 2. *Within each department, compute the highest salary by age. To smooth out abrupt variations, for each different value of age consider also the salary of the next youngest employee.*

The results for Query 2 are shown in Table 2.3 (for brevity, only results for the first department are shown). In this case, the MAX aggregate function is applied to collections of tuples generated by a process known as *partition composition*. During *partition composition* tuples sharing the same values in a list of attributes (termed *partition attributes*) are placed in the same collection. Because these collections result from *partitioning composition*, let us call them *partitions*. For Query 2 partition composition has been defined on *DepartmentId* resulting in two partitions. Because we want to generate an aggregate value for each entry in a partition, aggregate functions are not applied directly to partitions. Instead *sliding window composition* is

Name	*DepartmentId*	*Age*	*Salary*	*MAX(Salary)*
Praveen	1	24	45000	45000
John	1	28	42000	45000
Frank	1	50	80000	80000

TABLE 2.3. The Result of an Aggregate Query Using Partition Composition on *DepartmentId*, Sliding Window Composition on *Age*, and `MAX` Aggregate Function on *Salary*

performed on each partition. During sliding window composition, a *window frame* is placed around each tuple in the partition. A window frame is defined using a range of values (logical size) or a number of tuples (physical size), either leading or trailing (or both) each tuple in the partition. In the case of Query 2, the window frame was defined as "1 tuple trailing". This effectively selects a set of tuples on which an aggregate function is applied. For example, for the first tuple in the partition (*i.e.*, Praveen) there are no tuples trailing. Hence, the aggregate function is applied only to the current value of *Salary* (45000 in this case). For the second tuple in the partition (*i.e.*, John), the window frame selects the current and previous tuples in the partition. The `MAX` aggregate function is then applied to the set $\{45000, 42000\}$, resulting in the aggregate value 45000. For the third tuple in the partition, the aggregate function is applied to the set $\{42000, 80000\}$, yielding the aggregate value 80000. Note that in this case, we generate an aggregate value for every tuple in every partition of every group. If group composition is not used (such as in Query 2), the entire relation is considered as a single group.

To complement the aggregate functions, the SQL standard includes mechanisms for defining collection of tuples at these three levels (grouping, partitioning, and sliding window composition). The SQL-92 standard includes the `GROUP BY` clause to perform grouping composition. The need for partitioning and sliding window composition was later noted and the SQL:1999 standard includes the `WINDOW` clause to address both of these needs.

Aggregate	Evaluation	Evaluation with unique values
$COUNT_i(R)$	$\|R\|$	$\|\pi_{A_i}(R)\|$
$SUM_i(R)$	$\sum(\{r.A_i \| r \in R\})$	$\sum(\{r.A_i \| r \in \pi_{A_i}(R)\})$
$AVG_i(R)$	$SUM_i(R)/COUNT_i(R)$	$SUM_i(\pi_{A_i}(R))/COUNT_i(\pi_{A_i}(R))$
$MIN_i(R)$	$min(\{r.A_i \| r \in R\})$	$min(\{r.A_i \| r \in R\})$
$MAX_i(R)$	$max(\{r.A_i \| r \in R\})$	$max(\{r.A_i \| r \in R\})$

TABLE 2.4. Evaluation of the SQL-92 Aggregate Functions

2.2.1 Formal Definition of Traditional Aggregation

The aggregation functions defined by the SQL-92 standard can be evaluated as indicated in Table 2.4. Each of these functions operates over a virtual relation. This virtual relation is a collection of tuples defined by group composition, partition composition, and sliding window composition. Let us consider an aggregate query using group composition such as Query 1 presented in the previous section. In this query, tuples were first assigned to collections based on the value of their attribute *DepartmentId*. Then, an aggregate function (*e.g.*, `MAX`) was applied to each collection. The SQL standard describes how queries using group composition generates an aggregate value. We formalize the definition of these queries as follows.

Definition 1 (Aggregation using Group Composition). *Given an aggregation query on relation R and a select predicate SP, using group composition to define collections of tuples based on the values of attribute list A of R, the solution to the query can be generated as follows.*

Let S be the set of distinct values contained in the attribute list A. That is, $S = \pi_A(R)$. Every $s \in S$ partitions the value domain of A and generates groups of tuples from R as

$$G_{A,SP}(s,R) \;=\; \{r | r \in R \wedge r[A] = s[A] \wedge SP(r)\}\,. \tag{2.1}$$

Now, the solution to an aggregate query using the groups of tuples defined by S over

relation R, $S = \pi_A(R)$, is given by the expression

$$GAgg_{f_i,A,SP}(R) = \{s \circ f_i(G_{A,SP}(s,R))|s \in \pi_A(R)\}$$

where f_i is the aggregate function. This query produces a new relation whose schema is $A \cup Agg$.

Let us now consider an aggregate query using partition composition such as Query 2 presented in the previous section. As we have seen, this query generates collections of tuples based on the values of a list of attributes, but instead of generating a single aggregate value for each collection, an aggregate value for every tuple in the collection is generated. Therefore, there is a need to define a window to slide through all tuples in the collection. An aggregate function is then applied to the set of tuples covered by the window. Formally, an aggregate query using partition composition generates an aggregate value for each tuple in the relation as follows.

Definition 2 (Aggregation using Partition Composition). *Given an aggregation query on relation R and a select predicate SP, using the partition composition to define partitions of tuples based on the values of attribute list A of R and sliding window composition based on a single attribute B of R, the solution to the query can be generated as follows.*

Let the list of attributes A of R create a data partition P as defined by Equation 2.1. During sliding window composition, for each tuple p in P, a window frame is defined by the following expression [2].

$$WF_{precedes,follows,B}(p,P) = \{t|t \in P \wedge (p[B] - precedes) \leq t[B] \leq (p[B] + follows)\}$$

Now, the solution to an aggregate query using data partitions defined by A over relation R, window fames defined on attribute B of R, and a range on B defined by

[2]While the SQL standard contemplates the possibility of defining window frames by specifying its physical size, this implies having some ordering in the tuples. This is not possible using set algebra. Therefore we do not contemplate this possibility in our model.

precedes and follows, is given by the following expression.

$$WAgg_{f_i,A,SP,B,precedes,follows}(R) = \{p \circ f_i(WF_{precedes,follows,B}(p,P)) \mid p \in P_{A,SP}(s,R) \wedge s \in \pi_A(R)\}$$

The resulting relation has schema $R \cup Agg$.

We can evaluate Query 2 using this semantics. For this, we simply need to set the values of *precedes* and *follows* to 1 and 0, respectively. The sliding window is defined on attribute *Age* (*i.e.*, $B = Age$). Similarly, the list of partition attributes $A = \{DepartmentId\}$ and $f_i = MAX_{Salary}$.

2.2.2 Existing Approaches for Evaluating Aggregate Queries

A simple two-step algorithm was proposed by Epstein for evaluating aggregate queries [41]. To handle many aggregate functions in a query, the algorithm computes each of them separately and stores each result in a singleton relation, referring to that singleton relation when evaluating the rest of the query. A different approach employing program transformation methods was proposed by Freytag and Goodman to systematically generate efficient iterative programs for aggregate queries [46].

Recently, researchers have explored diverse aspects of the aggregation operation. Among them, **optimization**, for applications where performance is of utter importance [83, 128], **online aggregation**, where the user is aware of the progress made by the query processor and he/she is capable of stopping the query once an acceptable result have been achieved [57, 59], or **approximate** solutions, for applications where an exact solution is not required, and a fast *good* answer is preferred [23, 24, 49, 50]. These are techniques that can be applied to the computation of aggregate functions in general. We provide more detail whenever these techniques are presented as part of the existing approaches for evaluating temporal, spatial, or spatiotemporal aggregation.

2.2.3 Aggregation and OLAP

Typical OLAP queries aggregate data across several attributes (*i.e.*, columns) in a relation. The CUBE operator [53], for instance, was proposed as the n-dimensional generalization of the `GROUP-BY` clause in SQL. It computes `GROUP-BY`s corresponding to all possible combinations of a list of attributes. This implies finding the power set of all attributes in the relation, which is not a trivial task. Thus, solving aggregate queries in OLAP applications has inspired a considerable amount of research work. A general assumption in the CUBE operator is that the aggregate function being computed is distributive. Therefore, aggregate functions can be partially computed on disjoint subsets of data. By pre-computing the aggregated results of different subsets of data, the total processing time of a query can be drastically reduced. The final result can be obtained by properly merging these partial results [2, 30, 62].

Different columns in a data cube are usually called "dimensions,". However, some of these dimensions (*e.g.*, *CustomerId*) are defined over discrete domains which do not have a natural ordering among their values (customer 1000 cannot be considered "close" to customer 1001). In such cases, any ordering defined for the values in one of these columns is arbitrary. For this reason, we do not consider these "dimensions" as a special extent of the entities modeled by the database; instead, they can be regarded as explicit attributes that characterize a particular entity.

2.3 Temporal Aggregation

While a conventional database models the reality relevant to an enterprise as a single state, a temporal database is one that supports some aspect of time and keeps track of the different states of the database. Time-varying data is common, and applications that manage such data abound [14, 85, 133]. In a temporal database, the temporal data is modeled as a collection of line segments. These line segments have a begin time, an end time, one or more time-invariant attributes, and one or

more time-varying attributes. It is well known that database facts have at least two relevant temporal aspects. *Valid time* concerns when a fact was true in the modeled reality. *Transaction time*, on the other hand, concerns when a fact was current in the database. These two aspects are orthogonal, in that each could be independently recorded or not, and each has associated with it specific properties [14, 66, 114, 115]. All methods to date have focused on one time dimension only. However, most of them can be easily extended to handle either valid or transaction time.

Similar to the case for explicit attributes used in traditional aggregation, the temporal attributes of an object can be used to define collections of tuples on which to apply aggregate functions. These collections are defined by a process called *temporal grouping* [79], in which the time line is partitioned and tuples are grouped over these time partitions. Temporal aggregation, as studied in the literature, uses time granularities as the building blocks for temporal grouping. Different granularities of the time dimension can be used to define *temporal group composition*, *temporal partition composition*, and *temporal sliding widow composition.* Details on how this is achieved, along with illustrative examples, are presented in the following section.

2.3.1 Formal Definition of Temporal Aggregation

Computing temporal aggregates is a significantly more intricate problem than conventional aggregation because each database tuple is accompanied by a time interval during which its attribute values are valid. Consequently, the value of a tuple attribute affects the aggregate computation for all those instants included in the tuple's time interval.

In traditional databases, where only explicit attributes are of concern, aggregate functions are applied to collections of tuples that are defined by the different values in a list of explicit attributes. For the temporal extent of an object, collections of tuples can be defined based on time granularities (such characterization will allow the

approaches we discuss below to be classified).

A time domain is the set of primitive temporal entities used to define and interpret time-related concepts [40, 92]. Formally, a time domain is a totally ordered set of time points with the ordering relation $\leq$. A granularity creates a discrete image, in terms of *granules*, of the time domain. Portions of the time-domain are grouped into aggregations called *granules*. A granule is a subset of the time domain. A granularity is a mapping G from the integers to granules. Granularities are related in the sense that the granule in one granularity may be further aggregated to form larger granules belonging to a coarser granularity [15, 16, 40].

Temporal group composition is a mechanism that generates collections of tuples. A collection, termed a group, is formed by all tuples valid at the same time value at granularity G. An aggregate function can then be applied to each group. *Temporal partition composition* is used for handling queries that require aggregation at a finer level. Temporal partition composition defines collections of tuples, termed partitions, based on the distinct time values at granularity H (H *is finer than* G, denoted by $H \prec G$). However, aggregation functions are not applied to these partitions. Instead, *temporal sliding window composition* places window frames around each time value at granularity J ($J \prec H$) within these partitions. A window frame is defined by a time interval leading, trailing, or leading and trailing every time value in the partition. The aggregate function is applied to the set of tuples valid for the window frame around each time value within a partition.

The generation of collections of tuples based on some partition of the time domain have received several names in the research literature. In particular, we have encountered the terms *span grouping* and *instant grouping*. For span grouping, the time line is partitioned in pre-defined intervals such as year, month, or day [79]. Instant grouping, in the other hand is defined by the data [79, 88, 117]. These two are really special cases of temporal group composition. In the former, the granularity used is that of a year, month, day, etc. In the latter, the granularity used for temporal group

composition is the finest granularity supported by the temporal relation.

When computing temporal aggregation using group composition, the resulting relation is a time-varying relation defined at granularity G (*i.e.*, the granularity of the groups). Consider, for example, the following query.

Query 3. *Compute the monthly average salary of all employees.*

In this query, the time line is partitioned using fixed intervals (*i.e.*, months). Groups of tuples are defined by each temporal partition and the aggregation function is applied to each group. This kind of aggregation query is formally defined as follows.

Definition 3 (Aggregation using Temporal Group Composition). *Given a temporal relation R^T and a select predicate SP, let $\mathcal{T}(G, R^T) = \{\tau | \tau \in cast(r[vt], G) \wedge r \in R^T\}$, where $r[vt]$ gives the valid time of r, be the set of time values at granularity G, for which there is at least one tuple in the temporal relation R^T that is valid at that time value and at that granularity. Each time value $\tau \in \mathcal{T}(G, R^T)$ defines a collection of tuples in R^T based on a time partition as follows.*

$$\begin{aligned} P_{G,SP}(\tau, R^T) \;=\; \{t \mid \exists\, r \in R^T \wedge overlaps(cast(r[vt], G), \tau) \wedge SP(r) \\ \wedge\; (t[A_1, \ldots, A_n] = r[A_1, \ldots, A_n]) \\ \wedge\; (t[vt] = intersect(r[vt], \tau)\} \end{aligned} \tag{2.2}$$

The result of an aggregate query using temporal group composition with granularity G is given by the following expression.

$$GBAgg_{f_i,G,SP}(R^T) = \{\tau \circ f_i(P_{G,SP}(\tau, R^T)) | \tau \in \mathcal{T}(G, R^T)\}$$

In Equation 2.2, $r[A_1, \ldots, A_n]$ refers to the explicit attributes of tuple $r \in R^T$. Note that P, as defined by Equation 2.2, does not generate a strict partition of R^T such as the case of Equation 2.1. Instead, P is a subset of the rows in R^T. The temporal extent of the tuples in P has been narrowed to a single granule in granularity

G. Also note that, in order to provide a clean and simple notation, Equation 2.2 generates groups based on the implicit attribute *valid time*. This definition can be easily modified to account for *transaction time*.

For an aggregation using temporal partition composition, on the other hand, there are at least two granularities involved. The first (*i.e.*, coarser) granularity defines collections of tuples called partitions, whereas the second (*i.e.*, finer) granularity is used to define window frames during temporal sliding window composition. When temporal partition composition at granularity H is used in combination with temporal group composition at granularity G, $H \prec G$. For example, consider the following query requiring temporal partition composition and temporal window composition.

Query 4. *For every year, compute the moving average of salary of all employees with respect to the previous two months.*

In this case, a temporal partition is defined at the granularity level of year. Within each partition, tuples are grouped by month. All tuples valid at a particular month and the previous two months form a group on which the aggregate function is to be applied. The result of the query is an aggregate value for every time value at the granularity level of month. We can formally define this kind of query as follows.

Definition 4 (Aggregation using Temporal Partition Composition). *Given a temporal relation R^T and a select predicate SP, let us use $\mathcal{T}(H, R^T)$ and $P_{H,SP}(\tau, R^T)$ as before. Let J be a granularity such that $J \prec H$. For each $t \in \mathcal{T}(J, R^T)$, a window frame with respect to the time partition defined by granularity H is generated as*

$$WF_{H,SP,J,precedes,follows}(t, R^T) = \{r | r \in P_{H,SP}(cast(t, H), R^T) \wedge overlaps(cast(r[vt], J), [t - precedes, t + follows))\},$$

where precedes and follows are query arguments that define the aggregation group around each time value t within a window partition.

The result of an aggregate query on temporal relation R^T with temporal partitions at granularity H and window frames at granularity J with ranges defined by precedes and follows, is given here.

$$WAgg_{f_i,H,SP,J,precedes,follows}(R^T) = \{cast(t,H) \circ t \circ f_i(WF_{H,SP,J,precedes,follows}(t,R^T)) \mid t \in \mathcal{T}(J,R^T)\}$$

For Query 4, we have used the granularity level *year* to define partitions ($H = year$) and the granularity level *month* to define window frames ($J = month$). A window frame is defined by including the previous two temporal values at granularity level *month* (*i.e.*, $precedes = 2$) but no future values (*i.e.*, $follows = 0$). For all tuples valid within a particular window frame, the `AVG` aggregate function is applied.

2.3.2 Existing Approaches for Evaluating Temporal Aggregate Queries

Various algorithms have been proposed for processing temporal grouping and computing aggregation on a temporal relation. These algorithms can be classified based on the time when the aggregate value is computed. *Non-indexed evaluation* algorithms scan the temporal relation every time an aggregate query is issued. During this process, collections of tuples are generated based on temporal grouping and an aggregate function is applied to each collection. *Indexed evaluation* algorithms, on the other hand, pre-process aggregate values and store this information in a disk-based data structure. Instead of scanning the temporal relation when a query is issued, indexed evaluation algorithms use this data structure for answering an aggregation query.

Non-indexed Aggregation Evaluation

The earliest approach for evaluating temporal aggregation was proposed by Tuma [123]. He proposed a two-step algorithm. In the first step, the temporal relation is scanned

once to determine *constant intervals*. A constant interval is a period for which the temporal relation remains unchanged [117]. In the second step, the temporal relation is scanned again to apply the aggregate function on the groups of tuples defined by the constant intervals. Tuma's approach is based on Epstein's [41] algorithm for computing aggregation over explicit attributes using the `GROUP-BY` operator.

IO efficient algorithms for computing temporal aggregation were developed after Tuma's initial approach. These later methods require reading the temporal relation only once. A data structure (usually maintained in main memory) is created as tuples in the temporal relation are processed. The resulting data structure holds sufficient information to compute temporal aggregation.

The Aggregation Tree : Kline and Snodgrass [79] proposed an algorithm for computing temporal aggregation main memory-based data structure. The proposed algorithm was called *aggregation tree* because it builds a tree while scanning a temporal relation. After the tree has been built, the answer to the temporal aggregation query is obtained by traversing the tree in depth-first search. It should be noted that this tree is not balanced. Therefore, the order of tuples inserted into the aggregation tree affects its performance. If the tuples are sorted on the start time and inserted in that order, the aggregation tree would look more like a linked list, causing insertions to be slower than insertions into a balanced binary tree. For this reason, the worst case time to create an aggregation tree is $\mathcal{O}(n^2)$ for n tuples sorted in time. An even more serious limitation of the aggregation tree approach is that the entire tree must be kept in memory. Since the size of an aggregation tree is proportional to the number of distinct time stamps in the temporal relation, the size of the database the aggregation tree algorithm can deal with tends to be limited by the size of available memory and the number of distinct timestamps of tuples.

To minimize memory limitations, a variant of the aggregation tree, called *k-ordered aggregation tree*, was proposed by the same authors. The k-ordered aggregation tree

takes advantage of the *k-orderedness* of tuples to enable garbage collection of tree nodes, so that the memory requirements can be reduced significantly. However, the k-ordered aggregation tree approach assumes that the tuples in a table are ordered within a certain degree. Specifically, each tuple is at most k positions from its position in a totally ordered version of the table. This requirement is difficult to meet in a real database. Without *a priori* knowledge about a given table, the k-orderedness is expensive to measure, as it requires an external sort of the table. The worst case running time of the k-ordered aggregation tree algorithm is still $\mathcal{O}(n^2)$.

In an extension of his previous work, Kline [78] proposed to use a 2-3 tree, which is a balanced tree, to compute temporal aggregates. The leaf nodes of the tree store the time intervals of the aggregate results. Like the aggregation tree, this approach requires only one database scan. Note that, because it is a balanced tree, the running time is $\mathcal{O}(n \log n)$. However, its main limitation lies on the requirement that a database be initially sorted by start time. It has been shown that, for a randomly ordered database, the aggregation tree performs better than the 2-3 tree approach [78]. This is due to the preprocessing cost required by the 2-3 tree approach to sort the database.

The PA-tree: Kim *et al.* proposed an algorithm for computing temporal aggregation that is asymptotically better than the aggregation tree. The proposed method is based on the *point-based aggregation tree (PA-tree)* [76], which stores timestamps instead of intervals in an AVL tree. This approach requires one scan of the temporal relation for building the tree. Since the tree is balanced, the time complexity for building the tree is $\mathcal{O}(n \log n)$ rather than $\mathcal{O}(n^2)$ for the aggregation tree. In addition to timestamps, each node in the *PA-tree* stores either a single aggregate value for computational aggregates such as `COUNT`, `SUM`, and `AVG` aggregation, or a list of *value-length* pairs for selective aggregates such as `MIN` and `MAX` aggregation. Computing the algebraic aggregate functions is performed by doing an in-order traversal of the tree

and updating aggregate values by the amount indicated on each encountered node. Selective aggregate functions are computed by merging the lists of pairs associated to each tree node in similar way to the *skyline problem* [84].

Parallel Temporal Aggregation: Here, we discuss algorithms that have been developed for the parallel processing of temporal aggregation in large-scale databases. Ye and Keane proposed two approaches to parallelize the aggregation tree algorithm on a shared-memory architecture [130]. They propose to parallelize temporal aggregation queries that include `GROUP-BY` on explicit attributes. Each group defined by the grouping attribute is send to a processor where the temporal aggregation is computed locally.

Gendrano *et al.* have also developed several parallel algorithms [48] for computing temporal aggregates, specifically on a shared-nothing architecture, by parallelizing the aggregation tree algorithm. Gendrano *et al.* showed promising scale-up performance of the parallel algorithms through extensive empirical studies under various conditions. Nonetheless, all the aforementioned parallel algorithms inherit the same limitations from the aggregation tree algorithm, as the parallel algorithms were developed by parallelizing the aggregation tree. In particular, the size of the database those parallel algorithms can handle will be limited by the aggregate memory of participating processors.

All the non-indexed evaluation algorithms for temporal aggregation presented here address the same type of query. At a logical level, this type of query can be described as follows. First, for each time value τ at the finest granularity supported by the temporal relation, a collection of tuples is generated. The collection corresponding to the time value τ is formed by all tuples in the temporal relation valid during time τ. Second, an aggregate value is generated for each collection and the corresponding time value τ is annotated with this aggregate value. Finally, consecutive time values annotated with the same aggregate value are coalesced into a constant interval. That

is, a time interval for which the temporal aggregate value remains constant. Note that this type of query corresponds to temporal aggregation queries using temporal group composition as presented by Definition 3. The granularity used during group composition equals the finest granularity supported by the temporal relation.

Indexed Aggregation Evaluation

A more recent approach for evaluating temporal aggregation queries was proposed by Yang *et al.* They introduced the *SB-tree* [129] for incrementally computing temporal aggregates using a materialized view approach. This is a disk-based approach that computes temporal aggregates over a base relation that may gradually change by insertion and deletion. The SB-tree contains a hierarchy of intervals associated with partially computed aggregates. Aggregation over a given temporal interval is evaluated by performing a depth-first search on the tree and accumulating the partial aggregate values along the way. The SB-tree was developed for a data warehouse environment in which mostly insertions are expected. If deletion operations are expected, then `MIN` and `MAX` aggregation queries are not supported since these aggregate values cannot be incrementally maintained under deletions.

In addition to supporting temporal queries involving temporal group composition, the SB-tree supports queries that use a sliding window. These queries are called *cumulative aggregate queries* in Yang's *et al.* work [129]. For every time value τ at the finer granularity supported by the temporal relation, a cumulative aggregate query defines a window frame around τ using a time interval of length w preceding τ. All tuples valid during the interval $[\tau - w, \tau]$ form a collection for which an aggregate value is generated. Cumulative aggregate queries can be defined by our model using temporal partitioning and sliding windows. The granularity used for the sliding window definition should be the finest granularity supported by the temporal relation. The granularity used for the temporal partition definition should be so

coarse that all tuples in the temporal relation belong to the same collection.

One drawback of the SB-tree lies in the assumption that aggregate queries are always evaluated over the entire base relation. This is a clear disadvantage because aggregate queries usually specify a number of predicates to select the tuples on which temporal aggregation should be computed. The *multi-version SB-tree* (MVSB-tree) [136] was specifically designed to address this issue. It was proposed to deal with temporal aggregate queries coupled with range predicates on explicit attributes, termed *range temporal aggregates* [136]. The MVSB-tree is logically a series of SB-trees, one for each timestamp. Given a range on the values of one of the explicit attributes r, and a temporal interval i, the MSVB-tree computes the aggregate of all the tuples within r and valid during i as a series of additions and subtractions of values stored in the index. Because this is a form of pre-aggregation, only distributive aggregate functions can be evaluated by the MVSB-tree, in particular `SUM`, `COUNT`, and `AVG`.

The effectiveness of the MVSB-tree is limited by the size of its index, which can be larger than the database [122]. This limitation was overcome by Tao *et al.* [122] using an approach that computes an approximate solution to aggregate queries while maintaining only a small index. This approach is based, at a logical level, on a MVB-tree, but can be practically implemented using a B-tree and an R-tree. In particular, Tao *et al.* can approximately evaluate queries containing `COUNT` and `SUM` aggregate functions. Unfortunately, the approaches presented by Zhang *et al.* [136] and Tao etal [122] can only evaluate *non-sequenced* queries (*i.e.*, the query does not result in a temporal relation). This is because, once a set of tuples have been selected by a range predicate given on one of the explicit attributes and by a temporal predicate, the temporal information of the tuples is discarded.

One drawback of indexed evaluation algorithms is that they either assume that aggregate queries do not include predicates on the explicit attributes ([129]), or they assume that predicates are defined on a single attribute ([122] and [136]). Further-

more, the approaches presented by Tao *et al.* and Zhang *et al.* [122, 136] ignore the temporal characteristics of the tuples defined by the predicates once they have been selected. For this reason, we do not consider this approach a valid technique for evaluating temporal aggregates. A side effect of ignoring the temporal nature of data is the duplicate count problem, in which temporal objects may be counted more than once.

Another disadvantage of indexed evaluation algorithms is that they can only process certain types of aggregate functions. In particular, partial aggregation, or pre-aggregation, can only be used for distributive functions such as those included in the SQL-92 standard [53]. *Holistic* aggregate functions (*i.e.*, `MEDIAN`) cannot be combined with pre-aggregation. Therefore, queries involving holistic aggregate functions cannot be processed with indexed evaluation methods.

2.3.3 Aggregates on Data Streams

Data streams are ordered sequences of value points that are read/received in increasing order [6]. Because each value in a data stream is usually associated with a timestamp indicating either the time when the value was generated (*i.e.*, valid time) or the time when the value was received (*i.e.*, transaction time), data streams may be considered a special case of temporal data. Applications requiring the use of data streams are increasingly common and it is easy to find examples of data streams applications, such as network monitoring, security, telecommunications data management, web applications, manufacturing, transaction processing systems, and sensor networks [6, 38, 106, 135].

Because of the immense amount of data generated by the stream, it is extremely costly to store all data in such a way that it is readily available for answering queries. Instead, stream data is either discarded or archived after having been *looked at* just once. In consequence most applications only perform aggregate queries over data

streams. Summarized information about the data stream is often more important than retrieving specific entries with certain properties [98].

There are two models used for processing stream data [36, 135]. The *sliding window model* is used when only recent values in the data are of interest (*i.e.*, within the past w timestamps). The *complete (or infinite window) model* is used when all values in the stream are of interest. While stream data is a special case of temporal data, the complete model essentially ignores its temporal properties.

The sliding window model for processing stream data corresponds to temporal partition composition and temporal sliding window composition in our model. The sliding window composition is performed at the finest granularity supported by the timestamps on the values of the stream. The window frame is given by a trailing temporal interval of size w and there is no leading temporal interval. The temporal partition composition is such that the entire stream data creates only one collection of values. Methods developed to compute aggregation over data streams using the sliding window model include those by Datar *et al.* [36] and Zhang *et al.* [135]. Datar *et al.* present a method for computing approximate solutions for the `COUNT` and `SUM` aggregate functions. For this, they propose the use of *Exponential Histograms*, a data structure that can be incrementally maintained while preserving guarantees on the approximate solutions to `COUNT` and `SUM` aggregate queries. Zhang *et al.* [135] propose a mechanism for computing temporal aggregation on stream data based on a hierarchy of granularities. The main idea is to use different granularities to aggregate data depending on its age. Older data is aggregated at a coarser granularity whereas the most recent data is aggregate at the finest granularity. The most recent data is aggregated following the sliding window model. Established systems for stream data management have also adopted this approach for computing aggregate functions. One good example is Aurora [1], which is a model and an architecture for data stream management.

The complete model for processing data streams considers all values in the stream

read so far. Since data is not available at query time, only approximate solutions to aggregate queries are possible. For this, research work has turned to the maintenance of summarized information in the form of histograms [55, 98] or sketches [34, 38]. We do not provide further details on these methods because they ignore the temporal characteristics of data while evaluating aggregate queries.

2.3.4 Research Opportunities

Temporal aggregation queries can be evaluated by either non-indexed or indexed methods. Non-indexed evaluation methods require a scan of the base relation every time a query is issued. During this process, qualifying tuples are retrieved according to the select predicate and their aggregate information is maintained in a main-memory data structure. At the end of the scan, this data structure will provide a temporal relation with the aggregate values and their corresponding valid intervals. Unfortunately, all evaluation algorithms presented here rely on incrementally maintaining a main-memory data structure as the base relation is scanned. This approach will not work if the required data structure does not fit in the available memory.

Indexed evaluation methods do not require a scan of the base relation at query time. Instead, before any query is issued, a disk-based data structure that maintains pre-computed aggregate values is created. The evaluation algorithm uses this data structure to answer an aggregate query. The work of Yang and Widom [129], presented here, is the only indexed method that can evaluate a temporal aggregation query resulting in a temporal relation. The rest of the indexed evaluation algorithms evaluate a temporal aggregation query to a non-temporal relation. Because indexed evaluation algorithms rely on pre-aggregation, it is not possible for them to evaluate queries including non-distributive aggregate functions. Furthermore, they provide only limited (or null) support of selection predicates. If a query includes predicates not supported by the disk-based data structure, the query cannot be evaluated.

We have identified that temporal aggregation on large-scale databases is not properly supported by the methods presented in this chapter. Non-indexed methods are limited by the availability of main memory and can only be used to evaluate temporal aggregation queries on small databases. While indexed methods do not suffer as we scale-up the size of the database, they provide only limited support for select predicates. If queries with various kinds of select predicates are expected, indexed methods do not provide an effective solution. These drawbacks on the existing methods for evaluating temporal aggregate queries motivate our research work. In the following chapter, we present efficient algorithms for the evaluation of temporal aggregation on large-scale databases.

Chapter 3
Evaluating Temporal Aggregation Queries

As we have previously indicated, computing temporal aggregation is an expensive process. In particular, temporal group composition and temporal partition composition require determining the tuples that overlap each temporal instant, which is not a trivial task. Let us now provide an example of temporal aggregation to illustrate its time-varying nature. Table 3.1 shows a sample relation **t-employees** with two temporal attributes, which represent the beginning and ending of the valid-times of individual tuples. The resulting temporal aggregation of the maximum salary (along with the number of employees) is given in Table 3.2. Note that the aggregation results in a single time-evolving value.

Temporal aggregation can be processed in a sequential or parallel fashion. The parallel processing technology becomes even more attractive, as the size of data-intensive applications grows as evidenced in OLAP and data warehousing environments [25]. Although several sequential and parallel algorithms have been developed for the evaluation of temporal aggregation queries [48, 79, 117, 123, 130], they suffer from serious limitations such as the size of the temporal relation they can handle, restricted by

Name	Salary	Dept	Begin	End
Frank	46,000	Accounting	18	31
Praveen	45,000	Shipping	8	20
Sophia	35,000	Marketing	7	12
Sophia	38,000	Accounting	18	21

Table 3.1. T-Employees: A Sample Temporal Relation

COUNT	MAX	Begin	End
1	35,000	7	8
2	45,000	8	12
1	45,000	12	18
3	46,000	18	20
2	46,000	20	21
1	46,000	21	31

TABLE 3.2. The Evaluation of COUNT and MAX Temporal Aggregation Queries on T-employees

available memory, and the requirement of a priori knowledge about the orderedness of the input relation.

In this chapter, we propose a variety of algorithms for the evaluation of temporal aggregation queries that overcome major drawbacks of previous work. The main contributions of these algorithms can be summarized as follows.

- Two new algorithms proposed for small-scale aggregation do not require a priori knowledge about an input database, and they have improved both the worst-case and average-case processing time significantly.

- Another new algorithm proposed for large-scale aggregation that relies on a novel data partitioning scheme, so that it can deal with a database substantially larger than the size of available memory.

- Lastly, a parallel algorithm has been developed for shared-nothing architectures for large-scale aggregation. This solution achieves scalable performance by delivering nearly linear scale-up and speed-up, even at the presence of data skew.

It should be noted that the problem of computing temporal aggregates is different from the relational aggregation that can often be seen in the data warehousing environment. While data items in the data warehousing environment are envisioned

as points in their data domain, we deal with temporal data associated with time intervals of arbitrary lengths.

The rest of this chapter is organized as follows. In Sections 3.1, 3.2 and 3.3, we present improved algorithms for small-scale aggregation, and scalable solutions for large-scale aggregation based on data partitioning and parallel processing techniques. Section 3.4 presents the results from our experimental evaluation of the proposed sequential and parallel solutions. In Section 3.5, we discuss further extensions to the proposed algorithms. Finally, Section 3.6 summarizes the contributions of this chapter.

3.1 Improved Algorithms for Temporal Aggregation on Small-Scale Databases

In this section, we present two new algorithms for the evaluation of temporal aggregate queries. These algorithms are efficient alternatives to Kline's aggregation tree algorithm [79]. The aggregation tree is a binary tree similar to the segment tree by Bentley [10]. The segment tree is a static structure, which can be balanced for a given set of abscissae. However, there is no guarantee that the aggregation tree is always balanced. Because the aggregation tree is dynamically constructed as the tuples in a database are being scanned and inserted into the tree, the structure of the resulting aggregation tree depends on the order of tuples inserted. This fact may cause the worst case running time of $\mathcal{O}(n^2)$ for a database of n tuples, particularly when the tuples are ordered by their timestamp values. Such a quadratic complexity may be impractically costly for many database applications.

We have observed that the five most common aggregate functions (*i.e.*, those included in the SQL-92 standard) can be categorized into two groups, namely, `COUNT`, `SUM`, `MAX` in one group, and `MAX`, `MIN` in the other. For the latter group, there is more demand to keep track of attribute values of tuples. This observation has led

us to develop a different algorithm for each of the two groups of aggregate functions. The solution to the first group of aggregate functions, which we call the *balanced tree* algorithm, will be presented in Section 3.1.1. The main idea of this algorithm is that, by *giving up the notion of maintaining intervals* in the tree nodes, a tree can be balanced dynamically as tuples are being inserted. The solution to the second group is called the *merge-sort aggregation* algorithm, which is similar to the classical merge-sort algorithm [81]. This algorithm will be presented in Section 3.1.2. In this section, we assume that the main memory in our system is large enough to store the entire data structures required by each aggregation algorithm. In the rest of this chapter, we use the `COUNT` and `MAX` as the representatives of the two groups of aggregate functions, respectively.

3.1.1 Balanced Tree Algorithm for `COUNT` Aggregation

A relatively simple approach based on timestamp sorting can provide an effective mechanism for the evaluation of temporal `COUNT` aggregate queries. This approach starts by loading the entire temporal relation in main memory and extracting the timestamp values from the tuples. Each timestamp is then associated with a tag indicating whether the timestamp is the start time or the end time of a tuple. These timestamps and tags are then sorted in an increasing order of their values. See Figure 3.1 for a sorted list of timestamps and tags for the sample temporal relation given in Table 3.1.

The `COUNT` aggregate function is computed by scanning the sorted timestamps and tags in an increasing order. Getting started with a counter initialized to zero, the counter is incremented by one when a START tag is encountered, and it is decremented by one when an END tag is encountered. When more than one tags are associated with a timestamp, the counter is incremented by the number of START tags or decremented by the number of END tags. For example, in Figure 3.1, when

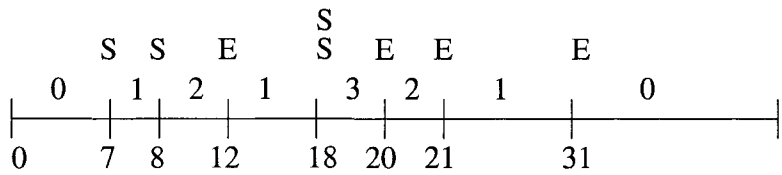


FIGURE 3.1. Example of COUNT Aggregation by Sorting Timestamps and Tags. Each timestamp is labeled to indicate whether a tuple starts or ends there. Timestamps are scanned in increasing order and the tags are used to determine the number of valid tuples for each interval.

the timestamp value 18 is encountered, the counter is incremented by two from 1 to 3 because there are two START tags associated with the timestamp. Apparently, the worst case processing time of this approach is $\mathcal{O}(n \log n)$, where n is the number of tuples in an input database.

In real world temporal databases, it may be the case that many tuples share the same timestamp values for their start times and end times. Unfortunately, given a temporal relation with n tuples, this timestamp-sort approach requires the same amount of memory and processing time regardless of the repeated timestamp values. That is, $2 \times n$ timestamps and $2 \times n$ tags are produced. Thus, we propose a *balanced tree* algorithm to further optimize performance for such databases with repeated timestamp values.

The motivation behind the balanced tree algorithm is that the sorted list of timestamps can be built even without loading an entire database into memory at once. Instead, timestamps can be sorted *incrementally* by inserting them into a balanced tree, as the tuples of an input database are being scanned. Each node of a balanced tree stores a timestamp, either a start time or an end time, but need not store a START/END tag. Instead of the tag, each node stores two counters: one storing the number of tuples starting at the timestamp and the other storing the number of

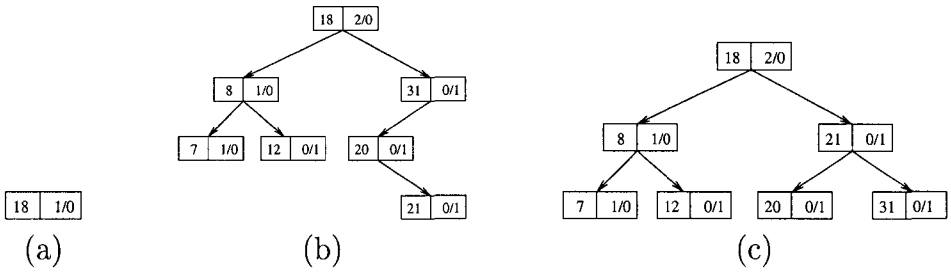


FIGURE 3.2. Building a Balanced Tree. After inserting the first node (a). Insertions can lead to a tree that is not balanced (b). A red-black tree is used to always have a balanced tree (c).

tuples ending at the timestamp. [1] Additionally, a color tag is stored in each node, as we use the *red-black* insertion algorithm [35] to keep the tree balanced dynamically.

Figure 3.2 shows the process of building a balanced tree for the sample **t-employees** temporal relation presented in Table 3.1. In the figure, we only show timestamps and counters, which are relevant to temporal aggregate computation. When the start time 18 of the first record is inserted into an empty tree, a new node is created for the timestamp, and then its start-counter and end-counter are set to one and zero, respectively. The resulting tree having a single node is shown in Figure 3.2(a). Figures 3.2(b) and (c) illustrate snapshots of the tree before and after the tree is balanced by the red-black insertion algorithm. We do not elaborate on the red-black insertion because it is not the focus of this work.

The balanced tree algorithm proceeds in two steps. During the first step, the tree is constructed while scanning the entire temporal relation. Whenever a tuple is read from the input database, the balanced tree is probed to see whether the start and end timestamps of the tuple are already in the tree. If the start (or end) timestamp is not found in the tree, then a new node is created and inserted into the tree. Otherwise, the start time (or end time) counter of a node that contains the timestamp is incremented

[1] For SUM aggregation, each node stores two variables: one storing the attribute value sum of the tuples starting at the timestamp and the other storing the attribute value sum of the tuples ending at the timestamp.

Algorithm 1: Balanced Tree Algorithm for `COUNT` Aggregation.

Input: R^T
// R^T *is a temporal relation.*
Output: The temporal evolution of the `COUNT` aggregate
set $\mathcal{T} \leftarrow$ an empty balanced tree;
for *each tuple* $t \in R^T$ **do**
  if *(*$t.start\_time = n.ts$ *for any node* n *in* $\mathcal{T}$*)* **then**
    $n.no\_starts++$;
  end
  else
    insert a new node n' (with $n'.ts = t.start\_time$) into $\mathcal{T}$;
  end
  if *(*$t.end\_time = n.ts$ *for any node* n *in* $\mathcal{T}$*)* **then**
    $n.no\_ends++$;
  end
  else
    insert a new node n' (with $n'.ts = t.end\_time$) into $\mathcal{T}$;
  end
end
set `COUNT` $\leftarrow 0$;
for *each node* n *in* $\mathcal{T}$ *traversed by in-order* **do**
  `COUNT` $+= n.no\_starts$;
  output $n.ts$ and `COUNT`;
  `COUNT` $-= n.no\_ends$;
end

by one without inserting a new node. In the second step, once the balanced tree has been built, the algorithm computes the aggregate values while performing an in-order traversal of the tree. Specifically, whenever a tree node is visited, the `COUNT` aggregate value is incremented by the start-counter value of the node and decremented by the end-counter value of the node. The proposed balanced tree algorithm is summarized in Algorithm 1.

By eliminating redundant timestamp values from the tree, the balanced tree algorithm reduces the memory requirements and tree traversal time substantially especially for a database with a small percentage of unique timestamps. The balanced tree stores information needed for temporal grouping and aggregation both in inter-

nal nodes and leaf nodes. Thus, the balanced tree algorithm uses only half the nodes required by the aggregation tree algorithm, which stores constant intervals only in leaf nodes.

3.1.2 Merge-Sort Algorithm for MAX Aggregation

While the balanced tree algorithm is a simple and efficient mechanism for evaluating temporal COUNT aggregate queries, it cannot be used for MAX aggregation. Since a balanced tree stores only unique timestamps and associated counters for COUNT aggregation, it is not possible to keep track of all the tuples that are valid at a given time instant with the information available in the tree. For example, in Figure 3.2(c), the root node shows that there exist two tuples whose start times are 18. However, the tree does not convey any information about the life spans of the tuples (*i.e.*, the exact end times of the two specific tuples). Unlike the COUNT aggregate function, it is impossible to compute the MAX aggregate function without knowing the exact life spans of the tuples in the database.

One could modify the balanced tree algorithm to compute the MAX aggregate function. This could be achieved by allowing repeated timestamp values in the tree and by using additional data structures such as dual heaps while traversing the tree. The dual heaps would be used to store the relevant attribute values (on which the MAX aggregation is performed) of "live" tuples and "dead" tuples, separately. While traversing the tree, the MAX aggregate value can be computed by comparing the two maximum values in both heaps and popping matched maximum values from the heaps. In fact, the dual heaps are used to keep track of the life spans of tuples, a necessary condition for computing the MAX aggregate function. However, with this modification, we lose all the benefits of using the balanced tree algorithm. That is, the tree will now need exactly two nodes per each tuple and we would not be able of reducing the memory requirements due to repeated timestamps. In addition, the

overhead for processing the heaps will be non-trivial.

Instead, we propose a *bottom-up* aggregation approach, which we call a *merge-sort aggregation* algorithm. Like the classical merge-sort algorithm based on the divide-and-conquer strategy, the merge-sort aggregation algorithm computes a larger (intermediate) aggregate result by merging two smaller (intermediate) aggregate results. The algorithm starts with merging tuples in pairs at the bottom and terminates when a final aggregate result is obtained at the top.

Formally, an intermediate aggregate can be defined as (T_k, M_k), where $T_k = \{t_0, t_1, \ldots, t_k\}$ and $M_k = \{m_1, m_2, \ldots, m_k\}$ for an integer $k \geq 1$. T_k is a set of $k+1$ unique timestamps in an increasing order ($t_0 < t_1 < \ldots < t_k$). M_k is a set of k attribute values, where $m_i\,(1 \leq i \leq k)$ is a maximum attribute value associated with a time interval $[t_{i-1}, t_i)$ if there exist at least one valid tuple in $[t_{1-1}, t_i)$. Otherwise, $m_i = nil$ for an empty interval. No two consecutive values in M_k are equal (*i.e.*, $m_i \neq m_{i+1}$ for any $i\,(1 \leq i \leq k-1)$). Each tuple t in an input database can be considered as a (T_1, M_1) with $T_1 = \{t.start\_time, t.end\_time\}$ and $M_1 = \{t.attribute\_value\}$.

Figure 3.3 illustrates the process of merging the tuples of the sample **t-employees** temporal relation given in Table 3.1. The tuples in this sample relation are described as four line segments in Figure 3.3(a). In the first step, the first two tuples in the **t-employees** relation are merged into an intermediate result $(\{8, 18, 31\}, \{45000, 46000\})$. Similarly, the last two tuples are merged into an intermediate result $(\{7, 12, 18, 21\}, \{35000, nil, 38000\})$. The result of the first step is shown in Figure 3.3(b). In a second step, the two intermediate results are merged together into the final aggregate result $(\{7, 8, 18, 31\}, \{35000, 45000, 46000\})$, as shown in Figure 3.3(c).

As an input temporal relation of n tuples is scanned, the merge-sort aggregation algorithm generates $\lceil n/2 \rceil$ first-step intermediate aggregates in memory. Then, the algorithm recursively merges the intermediate results until a final aggregate result is obtained. Thus, the worst case processing time of the algorithm is $\mathcal{O}(n \log n)$. As it is shown in Figure 3.3, the size of an intermediate result (T_k, M_k) can be smaller than

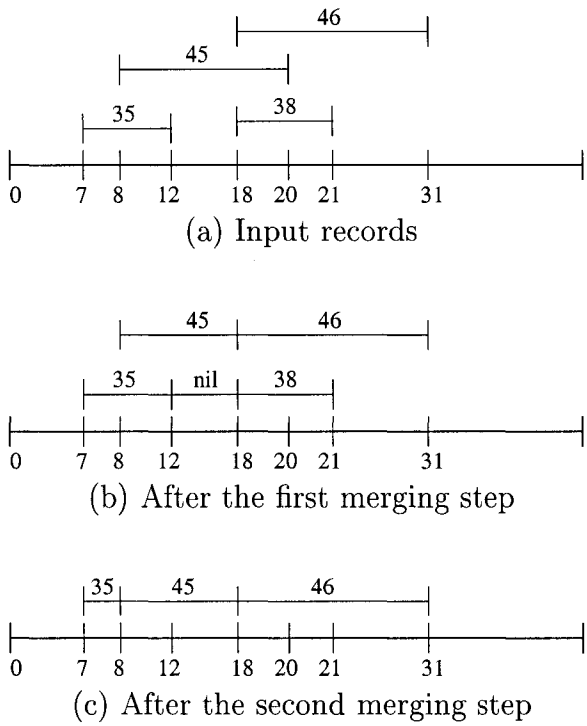


FIGURE 3.3. Example of Merging for `MAX` Aggregation. Tuples are considered segments that can be merged. After each merging step, the value assigned to a particular interval is the `MAX` of the merged values.

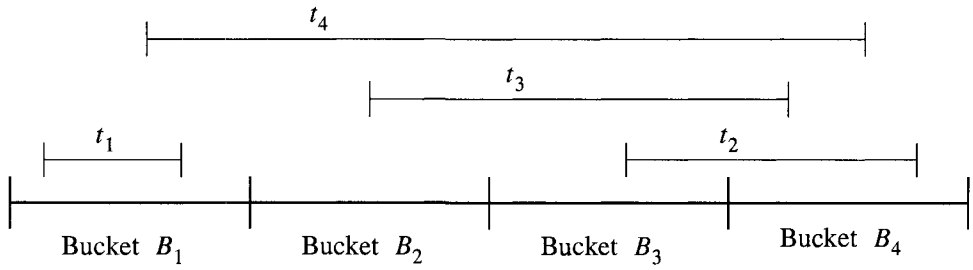
the tuples themselves covered by (T_k, M_k), because two consecutive intervals can be merged into a single interval if they share the same aggregate value. Thus, the amount of additional memory required for intermediate results is likely to be smaller than the size of an input database. Nonetheless, for evaluation temporal `COUNT` aggregate queries, the balanced tree will remain as the algorithm of choice. This is because the balanced tree algorithm will keep the memory requirement (*i.e.*, the number of tree nodes) down to the minimum by building a balanced tree incrementally and by removing repeated timestamps, and thereby minimizing its processing time.

3.2 Bucket Algorithm for Temporal Aggregation on Large-Scale Databases

In addition to the algorithms for evaluating aggregate queries on small-scale temporal relations proposed in the previous section, another major component of the work proposed in this chapter is the development of new techniques for the evaluation of temporal aggregate queries under the constraint of limited buffer space. Then, the size of databases we can deal with is not limited by the size of available memory. Additionally, it is crucial that temporal aggregation require only a constant number (say, two or three) of database scans, due to potentially huge amount of temporal data. It will be prohibitively costly to evaluate aggregate queries on a large-scale database, if the number of required database scans is not limited and is rather proportional to the size of database. For this reason, any method that requires more than a small constant number of database scans for evaluating an aggregate query is not acceptable.

In this section, we propose a new algorithm based on partitioning the tuples in a temporal relation into several buckets. Approaches based on data partitioning have been used for many important database operations such as the relational hash join algorithm. The idea of that latter join algorithm is to hash two joining relations on the join attribute, using the same hash function. Then, it is assured that tuples of one relation in a bucket can join only with tuples of the other relation in the same bucket. Thus once both relations are partitioned, the join operation can be performed by reading the relations just once, provided that enough memory is available to keep all the tuples of one relation in a bucket in memory. Assuming uniform distribution of data, it has been shown that the hash join algorithm requires three database scans if the number of buffer pages is larger than a square root of the number of disk pages in a smaller relation [37].

Although the idea of data partitioning appears promising for the relational hash

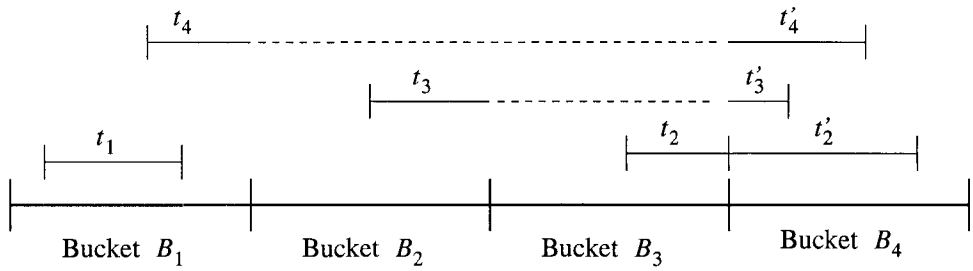


Bucket $B_1 = \{t_1, t_4\}$ Bucket $B_2 = \{t_3, t_4\}$
Bucket $B_3 = \{t_2, t_3, t_4\}$ Bucket $B_4 = \{t_2, t_3, t_4\}$

FIGURE 3.4. Time-line Partitioning and Assignment of Tuples into Buckets. A Naïve assignment of tuples can result in significant data replication.

join operation, it cannot be applied directly to evaluation of temporal aggregate queries. In our case, tuples associated with time intervals are not readily partitioned into temporally disjoint equivalent classes (*e.g.*, hash buckets). Because the time intervals of tuples may be of arbitrary length, some tuples may overlap with the intervals of more than one bucket. Such tuples must be checked with tuples in all the overlapping buckets. That is, there is no guarantee that temporal aggregate queries can be evaluated by reading the buckets only a constant number of times.

To circumvent this problem, one could simply replicate a data object into multiple buckets. This naïve approach can be best described by the example given in Figure 3.4. The time-line of a given temporal relation is partitioned into $\mathcal{N}_B$ disjoint intervals, where $\mathcal{N}_B$ is the number of buckets. If a tuple's life span is contained in the interval of a bucket, the tuple is assigned to the bucket. For example, in Figure 3.4, tuple t_1 will be assigned to bucket B_1 as t_1's life span is properly contained in that of bucket B_1. On the other hand, if a tuple's life span overlaps two or more intervals (say, k intervals), the tuple's life span is split into k pieces and these pieces may be assigned to k buckets. (It turns out that splitting a tuple into several does not impact the result of the aggregation.) In Figure 3.4, the life spans of tuples t_2, t_3 and t_4 overlap with 2, 3 and 4 buckets, respectively. Thus, tuple t_2 will be assigned to



Bucket $B_1 = \{t_1, t_4\}$ Bucket $B_2 = \{t_3\}$
Bucket $B_3 = \{t_2\}$ Bucket $B_4 = \{t'_2, t'_3, t'_4\}$

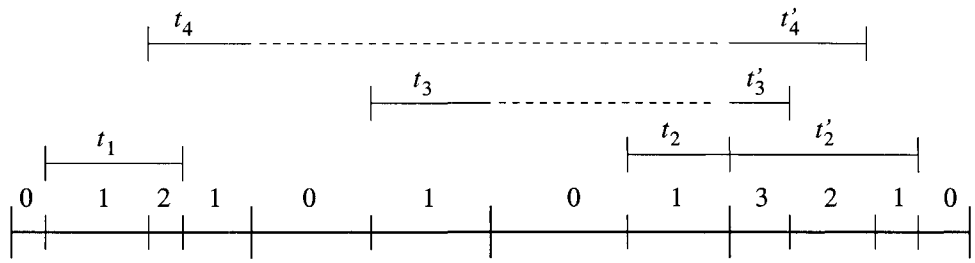
Meta Array for `count` aggregation $= \{0, 1, 2, 0\}$

FIGURE 3.5. Meta Array and Reduced Data Replication. Tuples that completely overlap a partition do not need to be replicated. We use the Meta Array to keep track of them. In the case of the `COUNT` aggregate function, the Meta Array just maintains a counter of the tuples overlapping each partition.

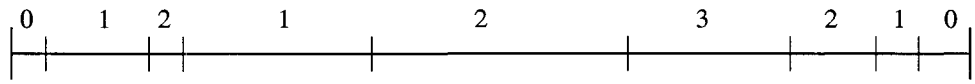
buckets B_3 and B_4, t_3 to buckets B_2, B_3 and B_4, and t_4 to buckets B_1, B_2, B_3 and B_4.

This process entails replicating tuples and may lead to considerable duplication of data, especially for long-lived tuples. To minimize duplication of tuples, we propose to assign each tuple solely to the buckets where the tuple's start and end timestamps lie. Suppose the life span of a tuple t overlaps buckets $B_i, B_{i+1}, \ldots, B_j (0 \leq i < j < \mathcal{N}_B)$. Then, the tuple t will be replicated only in the buckets B_i and B_j, but the intermediate buckets will not store the tuple t. Instead, a *meta array* is used to aggregate the information that the tuple t's life span overlaps the intermediate buckets $B_{i+1}, \ldots, B_{j-1}$. The size of a meta array is equal to the number of buckets. The i-th element of a meta array stores an aggregate value (*e.g.*, `COUNT`) for the i-th bucket.

For example, in Figure 3.5, the time interval of tuple t_3 spans over three buckets B_2, B_3, and B_4. Thus, t_3 is split into two segments (*i.e.*, t_3 and t'_3) with adjusted time intervals so that each segment can be properly contained in the interval of its corresponding bucket. (Solid lines in Figure 3.5 represent adjusted time intervals of



(a) After Aggregated Bucket by Bucket



(b) After Combined with the Meta Array

FIGURE 3.6. Steps of the Aggregation Based on Data Partitioning and Meta Array

split tuples.) Then, t_3 and t'_3 are assigned to two buckets B_2 and B_4, respectively; the third element of the meta array is incremented by one. In a similar way, t_4 and t'_4 are assigned to two buckets B_1 and B_4 respectively, and the second and third elements of the meta array are incremented by one. The resulting data partitioning and meta array are illustrated in Figure 3.5. Note that neither the first nor the last element of the meta array stores a valid aggregate value, as no tuple can have a life span longer than the time-line of an entire database.

Once all the tuples have been scanned and partitioned into buckets and the meta array has been created, the temporal aggregate function can be computed on each bucket independently. Figure 3.6(a) shows the partial results of the aggregation performed on each bucket. Then, each aggregate value stored in the meta array is combined with the aggregation results from each corresponding bucket (*e.g.*, simply by adding counts for `COUNT` aggregation). Lastly, the final aggregation results can be obtained by merging each pair of adjacent buckets at their boundaries if the two adjacent aggregate values are equal. Figure 3.6(b) shows the final aggregation results. The dotted vertical bars in the figure represent the merged bucket boundaries.

Algorithm 2 outlines our proposed temporal aggregation approach based on data partitioning. In the algorithm description, it is assumed that the entire time-line of a table is partitioned into $\mathcal{N}_B$ disjoint intervals of equal length. Each of these intervals is associated with a bucket. Note that any small-scale aggregation algorithm proposed in the previous section can be used to aggregate each individual bucket.

Algorithm 2: Bucket Algorithm based on Temporal Data Partitioning

Input: R^T
// R^T *is a temporal relation.*
Output: The temporal evolution of the COUNT aggregate
set $\mathcal{I}_B \leftarrow$ time interval for each bucket $((\mathcal{T}_{max} - \mathcal{T}_{min})/\mathcal{N}_B)$;
for *each tuple* $t \in R^T$ **do**
  set $start\_bucket \leftarrow (t.start\_time - \mathcal{T}_{min})/\mathcal{I}_B$;
  set $end\_bucket \leftarrow (t.end\_time - \mathcal{T}_{min})/\mathcal{I}_B$;
  insert t into a bucket B_{start_bucket};
  if $(start\_bucket \neq end\_bucket)$ **then**
    insert t' into a bucket B_{end_bucket};
  end
  for $(i = start\_bucket + 1$ ***to*** $end\_bucket - 1)$ **do**
    update $meta\_array[i]$;
  end
end
for $(i = 0$ ***to*** $\mathcal{N}_B - 1)$ **do**
  perform temporal aggregation on the bucket B_i;
  combine the scalar value of $meta\_array[i]$ to the bucket B_i;
  merge the bucket boundary with B_{i-1} as needed;
end

Provided that the number of tuples in each bucket is small enough such that there is sufficient main memory available to use one of the small-scale aggregation algorithms, the temporal aggregate function can be computed by reading each bucket just once. Therefore, overall this approach requires three database scans (*i.e.*, two reads and one write) to evaluate temporal aggregate queries. Considering the data replication for the tuples overlapped with multiple buckets, the database access requirement of this approach is likely to increase to some extent depending on various factors such as the life spans of tuples and the number of buckets used. Even in the

worst case, however, the size of a given table can increase only up to twice its original size by replicating each tuple in the table into two buckets. Thus, the database access requirement of this approach is still bounded to a small constant number of scans. We will show the performance impact of data replication in Section 3.4.

3.3 Parallel Bucket Algorithm

Parallel processing for database applications typically involves partitioning of data, followed by allocation of the partitions to a set of processors. Then, the processors perform operations on the partitioned data in parallel, achieving speed-up in query processing times. Among the various architectures that have been proposed for parallel database systems, a *shared-nothing* architecture [119] has made it an attractive choice for large-scale database applications due to its high potential for scalability. By scalability we mean the capability of delivering an increase in performance proportional to an increase in the number of participating processors.

In a shared-nothing architecture, each processor owns local memory and secondary storage units, and processors communicate to each other by message passing. Initial data placement can be either centralized or distributed across multiple processors. For most of the parallel database operations, however, some of the data may have to be redistributed among the processors that actually participate in the operations. We assume that resulting aggregate values will remain in local storage units of the participating processors without collecting the results on a special coordinator processor. In this way, these aggregate values can be used as intermediate data for the next phase of parallel query processing.

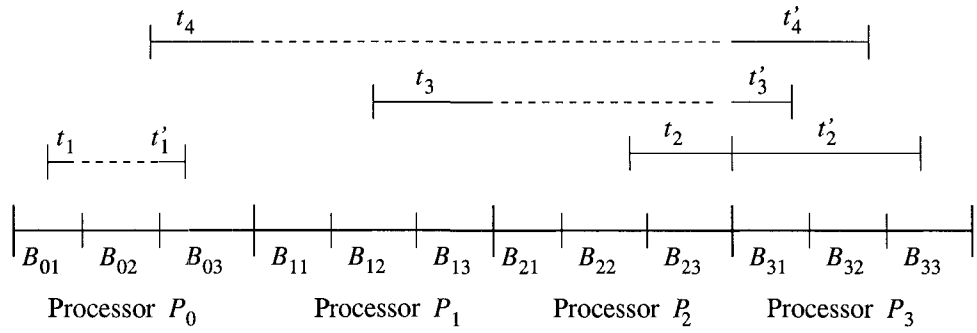
As we pointed out in Section 2.3.2, most of the previous attempts to develop scalable methods for the evaluation of large-scale temporal aggregate functions were based on parallelizing the aggregation tree algorithm. For the reason, those approaches inherit all the limitations the aggregation tree algorithm has. Specifically,

these approaches will suffer from $\mathcal{O}(n^2)$ worst-case running time and tight limitations on the size of the database they can deal with.

In this section, we propose a new parallel temporal aggregation algorithm based on the bucket algorithm (Algorithm 2) presented in the previous section. It is relatively straightforward to parallelize the bucket algorithm by distributing buckets across participating processors. The time-line of a given temporal database is partitioned into $\mathcal{P}$ disjoint intervals, where $\mathcal{P}$ is the number of processors. Then, on each processor, the time-line of the processor is again partitioned into $\mathcal{N}_B$ disjoint intervals. However, distributing the buckets is not sufficient to properly evaluate temporal aggregate queries, because the construction of meta arrays must also be processed in parallel in an efficient way.

We propose to use a *local meta array* and a *global meta array* on each processor for tuples whose life spans overlap time-lines of multiple local buckets and multiple processors, respectively. Specifically, if the life span of a tuple t overlaps the k-th bucket ($B_{P_i k}$) of processor P_i through the l-th bucket ($B_{P_j l}$) of processor P_j, the tuple t will be replicated only in $B_{P_i k}$ and $B_{P_j l}$. Then, a local meta array of P_i is used to aggregate the information that the tuple t's life span overlaps the intermediate buckets $B_{P_i k+1}, \ldots, B_{P_i \mathcal{N}_B}$, and so is a local meta array of P_j for $B_{P_i 1}, \ldots, B_{P_i l-1}$. Finally, a global meta array is updated on a processor that owns the tuple t to inform the intermediate processors $P_{i+1}, \ldots, P_{j-1}$ of the existence of the tuple t overlapped with their time-lines. The size of a global meta array is equal to the number of processors. The i-th element of a global meta array stores an aggregate value (*e.g.*, `COUNT`) for the i-th processor. Local meta arrays are identical with the ones used for the sequential bucket algorithm. Each processor computes its own global and local meta arrays independently.

In Figure 3.7, for example, suppose that tuples $t_1, \ldots, t_4$ are initially stored on a processor P_0, and four processors $P_0, \ldots, P_3$ participate in the evaluation of a temporal `COUNT` aggregate query. Since the time interval of t_3 spans over three remote processors



Bucket $B_{01} = \{t_1\}$ Bucket $B_{02} = \{\}$ Bucket $B_{03} = \{t'_1, t_4\}$

Tuples sent from P_0 to $P_1 = \{t_3\}$
Tuples sent from P_0 to $P_2 = \{t_2\}$
Tuples sent from P_0 to $P_3 = \{t'_2, t'_3, t'_4\}$
P_0's Local Meta Array for `count` aggregation $= \{0, 1, 0\}$
P_0's Global Meta Array for `count` aggregation $= \{0, 1, 2, 0\}$

FIGURE 3.7. Data Distribution and Meta Arrays for P_0's Local Data

P_1, P_2 and P_3, t_3 is split into two segments t_3 and t'_3, which are then sent to the processors P_1 and P_3, respectively. Then, the third element of the global meta array of P_0 is incremented by one. In a similar way, t_4 is assigned to P_0's local bucket B_{03} and t'_4 is sent to processor P_3; the second and third elements of P_0's global meta array are incremented by one. Figure 3.7 shows the resulting data distribution across P_0's local buckets, data shipping to other processors, and P_0's local and global meta arrays. Note that there may be some tuples sent from other processors to P_0, but they are not shown in Figure 3.7.

The proposed approach for the parallel evaluation of temporal aggregate queries is summarized in Algorithm 3. In the algorithm description, it is assumed that the entire time-line of a table is partitioned into $\mathcal{N}_B \times \mathcal{P}$ disjoint intervals of an equal length, each of which is associated with a bucket, and the buckets are distributed across $\mathcal{P}$ processors by range partitioning so that each processor is assigned $\mathcal{N}_B$ consecutive buckets. This range partitioning scheme obviously minimizes the size of a global meta array in a way that only one array element is required per processor. Since each

Algorithm 3: Parallel Bucket Algorithm based on Temporal Data Partitioning

Input: R^T
// R^T *is a temporal relation.*
Output: The temporal evolution of the COUNT aggregate on R^T
set $\mathcal{P} \leftarrow$ number of participating processors;
set $\mathcal{I}_B \leftarrow$ time interval for each bucket $((\mathcal{T}_{max} - \mathcal{T}_{min})/(\mathcal{N}_B \times \mathcal{P}))$;
set *this_proc* $\leftarrow$ a local processor id $(0 \leq this\_proc < \mathcal{P})$;
for *each tuple* $t \in R^T$ *in a local partition or from a remote processor* **do**
  set $start\_proc \leftarrow (t.start\_time - \mathcal{T}_{min})/(\mathcal{I}_B \times \mathcal{P})$;
  set $end\_proc \leftarrow (t.end\_time - \mathcal{T}_{min})/(\mathcal{I}_B \times \mathcal{P})$;
  if $(start\_proc \neq this\_proc)$ **then**
    send t to a processor P_{start_proc};
  end
  if $(end\_proc \neq this\_proc)$ **then**
    send t' to a processor P_{end_proc};
  end
  for $(i = start\_proc + 1$ ***to*** $end\_proc - 1)$ **do**
    update $global\_meta\_array[i]$;
  end
  insert t into one or two local buckets as in **Algorithm 2**;
  update *local_meta_array* as in **Algorithm 2**;
end
Globally combine the *global_meta_array* wrt. an aggregate operator *op*;
for $(i = 0$ ***to*** $\mathcal{N}_B - 1)$ **do**
  $local\_meta\_array[i] \leftarrow op(local\_meta\_array[i], global\_meta\_array[this\_proc])$;
  perform temporal aggregation on the bucket B_i with $local\_meta\_array[i]$
  as in **Algorithm 2**;
end

processor computes a global meta array independently only for its local data, all the $\mathcal{P}$ processors need to communicate each other to compute a final global meta array for an entire database with respect to a given operator *op*. The operator *op* is determined by a kind of aggregate operation. For example, *op* will be the *addition* operator for `COUNT` aggregation and the *maximum* operator for `MAX` aggregation. Such collective communication for computing a final global meta array can be implemented efficiently on most parallel computers and networks of workstations [7]. Thus the overhead for combining global meta arrays is expected to be negligible because the volume of

communication is only $\mathcal{P}$ words per processor.

3.3.1 Handling Data Skew

When describing the sequential and parallel bucket algorithms, we have assumed that the time-line of a temporal database is partitioned into $\mathcal{N}_B$ (or $\mathcal{N}_B \times \mathcal{P}$) disjoint buckets of equal length. Evidently this simple data partitioning scheme will not work for a database with skewed distribution of temporal attribute values, suffering from load imbalance. An interesting question that arises with respect to applying those bucket algorithms is how to ensure that (1) each processor receives about the same number of tuples to achieve load balance among processors, and (2) each bucket receives about the same number of tuples to minimize the local computational requirement (refer to the discussion in Section 3.4.3).

We address the issue of data partitioning for skewed data by determining the size of each bucket *adaptively* utilizing the selectivity estimation mechanism provided by most database systems. We assume that the frequency distribution of temporal attribute values is provided in an equi-depth histogram [95], then the partitioned time-line of a processor P_i is determined as the interval $[h[i \times \mathcal{N}_H/\mathcal{P}], h[(i+1) \times \mathcal{N}_H/\mathcal{P}])$, where $h[0..\mathcal{N}_H]$ is a set of temporal attribute values of the equi-depth histogram. That is, we assign an equal number of bins to each processor and partition the time line based on the temporal limits of these bins. Since each bin contains the same number of tuples (actually, their start or end timestamps), the workload across processors will be balanced. The boundaries for local buckets can be computed independently by each processor in a similar fashion. While this approach provided good results for us, the meta array can be used with any data partitioning scheme without any modifications. One alternative approach for partitioning is the method proposed by Soo *et al.* [118].

3.4 Performance Evaluation

In this section, we evaluate the proposed algorithms empirically and compare with the previous work. We chose the `COUNT` and `MAX` temporal aggregate queries as our benchmarks and carried out experiments under various operational conditions that may affect the performance of the algorithms. In particular, we focus on the performance gains by the proposed algorithms for small-scale aggregation, and on the scalability of the sequential and parallel bucket algorithms.

3.4.1 Experimental Settings

Testing and benchmarks were performed on a cluster of 64 Intel Pentium workstations with 200 MHz clock rate. Each workstation has 128 MBytes of memory and 2 or 4 GBytes of disk storage with Ultra-wide SCSI interface, and runs on Linux kernel version 2.0.30. The workstations are connected by a 100 Mbps switched Ethernet network. The switch can handle an aggregate bandwidth of 2.4 Gbps in an all-to-all type communication. That is, up to 24 nodes can use the switch at full rate before it saturates. For message passing between the Pentium workstations, we used the LAM implementation of the MPI communication standard [19]. With the LAM message passing package on the Pentium cluster, we observed an average communication latency of 790 microseconds and an average transfer rate of about 5 Mbytes/second. Note that this is relatively high latency and low transfer rate compared with parallel computers equipped with high performance switches such as IBM SP-2 parallel systems.[2]

For both sequential and parallel implementations, the same buffer size of 4 Kbytes was used for disk IO and message passing. Non-blocking message passing primitives were used in an attempt to minimize communication overhead by allowing

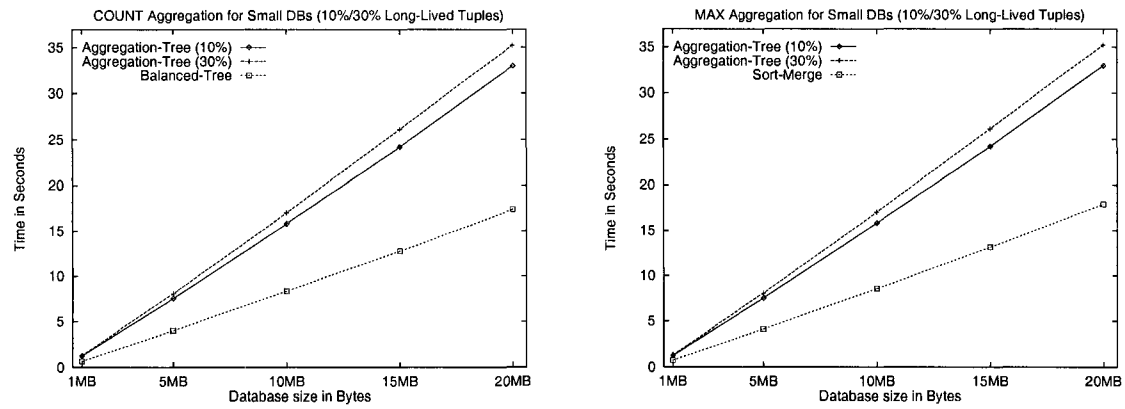
[2]On a SP-2 system with a proprietary MPI implementation `mpif`, we observed an average communication latency of 55 microseconds and an average transfer rate of about 35 Mbytes/second.

inter-processor communication to be overlapped with local computation and disk IO. Throughout the experiments, we measured elapsed times including disk access time and communication overhead. For accurate measurement, we averaged elapsed times from multiple runs after eliminating extreme cases. Additionally, we avoided the system cache effects for disk accesses by loading irrelevant data into the entire memory between consecutive runs of our experiments.

We generated synthetic data in the same way as in Kline's study [79]. Each database has a time-line of one million temporal instants. We considered two basic life spans for tuples: short-lived and long-lived. The life span of a short-lived tuple was determined randomly between one and 1,000 instants; the life span of a long-lived tuple was determined randomly between 200,000 and 800,000 instants, namely, between 20 and 80 percent of the time-line of a database. In most of our experiments, the population of long-lived tuples was fixed at 10 percent or 30 percent of the total number of tuples in the database. The start times of tuples were uniformly distributed over the time-line of a database. Each tuple was 20 bytes wide, including two temporal attributes (start time and end time) and other non-temporal attributes as well. Synthetically-generated databases used in our experiments were not sorted by any temporal attribute unless stated otherwise.

3.4.2 Small-Scale Aggregation

The first set of experiments were conducted on a single processor using relatively small databases between 1 MBytes and 20 MBytes so that all the required data structures can fit in available memory. We should recall that the algorithms proposed in Section 3.1, as well as the aggregation tree algorithm and its variations, require that the entire data structures be kept in main memory. In this section, we used the *balanced tree* algorithm for evaluating temporal `COUNT` aggregate queries, and the *merge-sort aggregation* algorithm for evaluating temporal `MAX` aggregate queries.

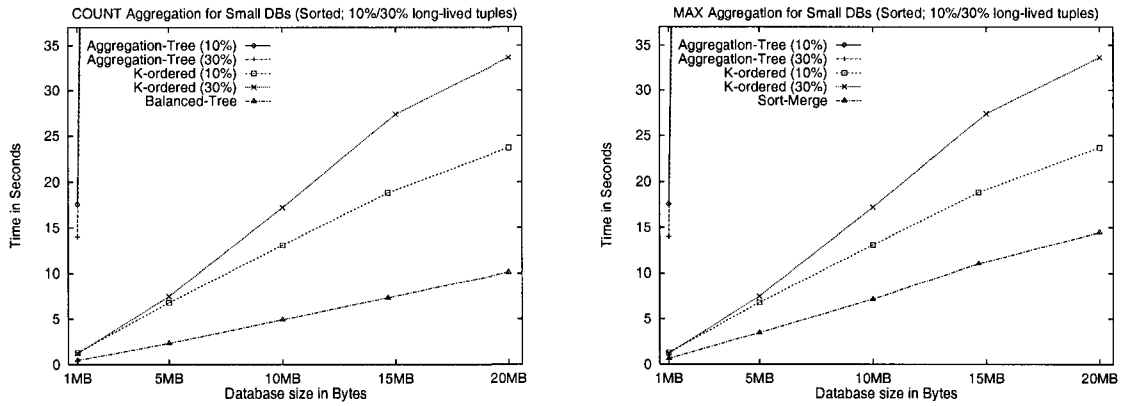


(a) COUNT Aggregation (b) MAX Aggregation

FIGURE 3.8. Elapsed Time for the Evaluation of Temporal Aggregation Queries on Small-scale Unsorted Databases. The Aggregation Tree is compared to the Balanced Tree (a) and to the Sort-Merge (b).

Figure 3.8(a) compares the balanced tree and aggregation tree algorithms for COUNT aggregation; Figure 3.8(b) compares the merge-sort and aggregation tree algorithms for MAX aggregation. The proposed balanced tree and merge-sort aggregation algorithms consistently performed about two times faster than the aggregation tree algorithm for COUNT and MAX aggregations, respectively. While the aggregation tree took more time to aggregate a database with higher percentage of long-lived tuples, the processing times of the two proposed algorithms remained constant for different percentage of long-lived tuples. Note that the performance of the aggregation tree algorithm remains unchanged for COUNT and MAX aggregations, since the algorithm works essentially in the same way for both aggregate functions.

In Figure 3.9(a) and (b), the tuples in input databases were sorted by their start time, where we expected the worst-case performance from the aggregation tree algorithm. The processing times of the aggregation tree were several orders of magnitude slower than the two proposed algorithms, and were plotted as almost vertical lines in the figures. Thus, we compared with the k-ordered aggregation tree algorithm (with $k = 1$) instead. The proposed algorithm still performed two to three times faster



(a) COUNT Aggregation (b) MAX Aggregation

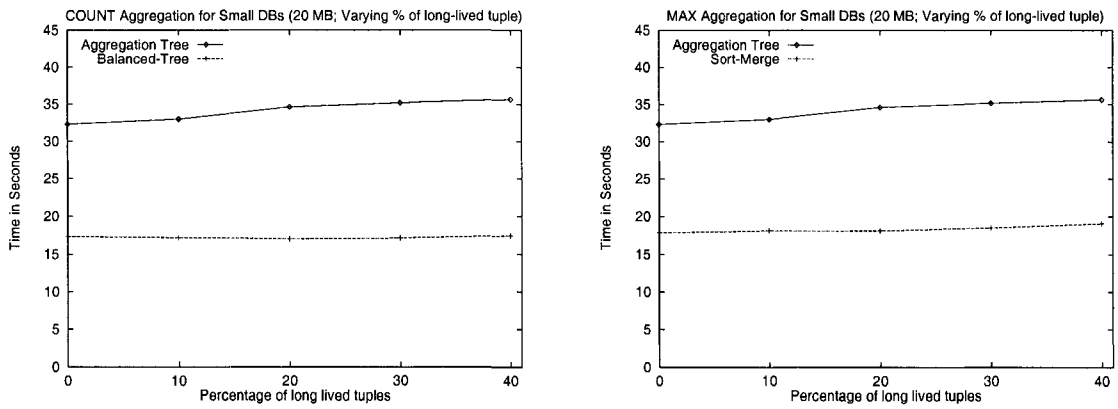
FIGURE 3.9. Elapsed Time for the Evaluation of Temporal Aggregation Queries on Small-scale Sorted Databases. The Aggregation tree performs poorly on sorted data and its performance plot shows as a vertical line. The Balanced tree (a) and Sort-Merge (b) perform better than the K-ordered Aggregation tree.

than the k-ordered aggregation tree algorithm.

In summary, the proposed algorithms outperformed the aggregation tree and k-ordered aggregation tree consistently by a significant margin. The k-ordered aggregation tree requires a priori knowledge about the orderedness of databases, whereas the proposed algorithms do not (such knowledge will help reduce the processing time of the merge-sort algorithm, but it is not required). The performance of the proposed algorithms was not affected by the percentage of long-lived tuples, as Figure 3.10(a) and (b) show the processing times measured on databases (20 MB) with a varying percentage of long-lived tuples.

3.4.3 Bucket Algorithm for Large-Scale Aggregation

Despite the fact that the balanced tree and merge-sort aggregation algorithms were designed for two different groups of aggregate operations, both algorithms showed almost identical performance behaviors in the previous experiments. Thus, for the rest of this section, we present experimental results only for COUNT aggregation.



(a) COUNT Aggregation (b) MAX Aggregation

FIGURE 3.10. Elapsed Time for the Evaluation of Temporal Aggregation Queries on Small-scale Databases with Varying Percentage of Long-lived Tuples. The Balanced Tree (a) and the Sort-Merge (b) perform better than the Aggregation Tree and are not sensitive to the percentage of long-lived tuples in the database.

The second set of experiments was designed to evaluate the performance of the *bucket* algorithm proposed in Section 3.2 using a single processor. First, we evaluated aggregate queries with and without using data partitioning for small databases, so that we could measure the overhead of data partitioning. The balanced tree algorithm was used for COUNT aggregation. In Figure 3.11, we used 64 buckets irrespective of database sizes, which was large enough to demonstrate the overhead of data partitioning. Compared to the balanced tree algorithm without data partitioning, we observed about 10 to 30 percent increase in processing time of the bucket algorithm. Despite the additional overhead, however, the bucket algorithm still outperformed the aggregation tree algorithm significantly (compare Figure 3.8(a) and Figure 3.11).

For small databases, the observed overhead due to data partitioning is expected to be smaller than the overhead for large databases. Because all the buckets might remain in memory even after they were written to disk, for the next step of aggregating individual buckets, the cached buckets would be used instead of the disk copies. Also note that performance of the bucket algorithm is affected by the percentage of long-

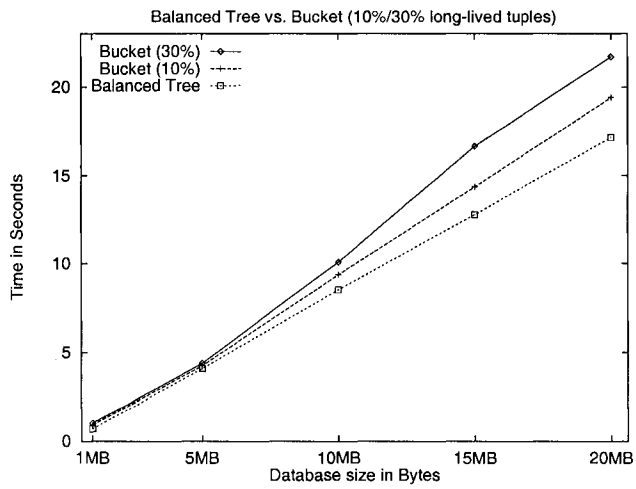


FIGURE 3.11. Overhead of the Bucket Algorithm Compared to the Balanced Tree on the Elapsed Time for the Evaluation of Temporal Aggregation Queries (`COUNT`)

lived tuples. The reason appears quite obvious because long-lived tuples are more likely to be replicated than short-lived tuples, leading to increased computation time and disk access time.

From the experiments, we have noticed that performance of the bucket algorithm is affected by the number of buckets used for data partitioning. More interestingly, there seemed to exist local optimum values, which were determined by database sizes. For example, in Figure 3.12, three, eight and fifteen were the optimal bucket numbers for a database of 100 MBytes, 200 MBytes and 400 MBytes with 10 percent of long-lived tuples, respectively. Our conjecture is that this is caused by two contravening performance effects from data partitioning. First, since the computational complexity of the balanced tree algorithm is higher than linear ($\mathcal{O}(n \log n)$), the overall computational complexity will be reduced by data partitioning. Specifically, the cost of a balanced tree construction is reduced from $\mathcal{O}(n \log n)$ down to $\mathcal{O}(n \log n - n \log \mathcal{N}_B)$, where n is the number of tuples in the relation and $\mathcal{N}_B$ is the number of buckets. Second, the more buckets are used for data partitioning, the more tuples are likely to be replicated, which will in turn increase the cost of disk accesses. We acknowledge that

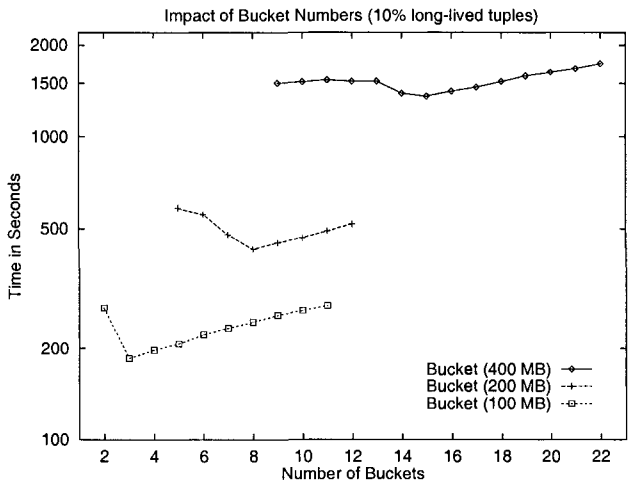


FIGURE 3.12. Impact of the Number of Buckets on the Elapsed Time for the Evaluation of Temporal Aggregation Queries. The Optimal Number of Buckets Depends on the Size of the Database.

this issue should be addressed more carefully and we refer the reader to Section 3.5.3 for a detailed discussion.

Figure 3.13 shows processing times of the bucket algorithm for databases of size from 20 MBytes up to 1 GBytes. The number of buckets used for data partitioning was 2, 8, 16, 24, 32 and 40 for 20 MBytes, 200 MBytes, 400 MBytes, 600 MBytes, 800 MBytes and 1 GBytes databases, respectively. We chose these number of buckets based on the observed performance shown in Figure 3.12. Since each of these databases is too large to fit in memory (with an exception of a 20 MByte database), none of the small-scale aggregation algorithms could be used for this experiment. The results shown in Figure 3.13 demonstrate that the proposed bucket algorithm can compute temporal aggregates for databases substantially larger than the size of available memory. However, it should be noted that the processing time of the algorithm grows faster than linear as the size of a database increases. This clearly motivates the need of scalable solutions such as the parallel bucket algorithm we proposed in Section 3.3.

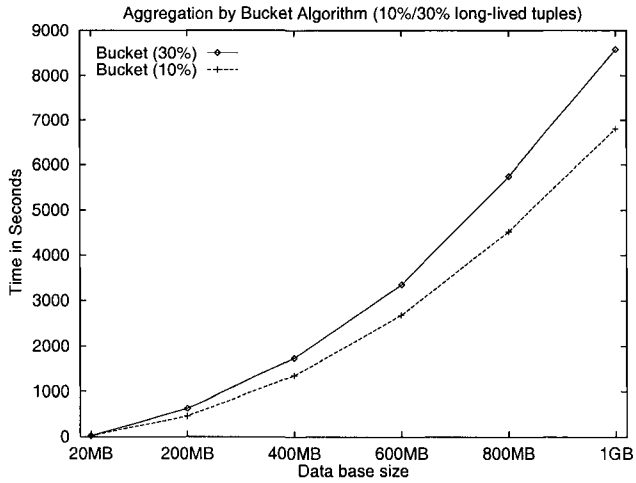


FIGURE 3.13. Scale-up of the Bucket Algorithm for the Evaluation of Temporal Aggregation Queries

3.4.4 Parallel Algorithm for Large-Scale Aggregation

The third set of experiments was designed to evaluate the scalability of the parallel bucket algorithm proposed in Section 3.3. For all the experiments presented in this section, input databases were distributed across participating processors by round-robin partitioning on a non-temporal attribute. By choosing such a non-temporal partitioning scheme for initial data placement, we can effectively eliminate any potential advantage that the parallel bucket algorithm can exploit for better performance. On the other hand, range partitioning on a temporal attribute would be the most favorable data placement for the parallel bucket algorithm, because the number of tuples to be shipped to remote processors could be minimized and thereby reducing communication overhead.

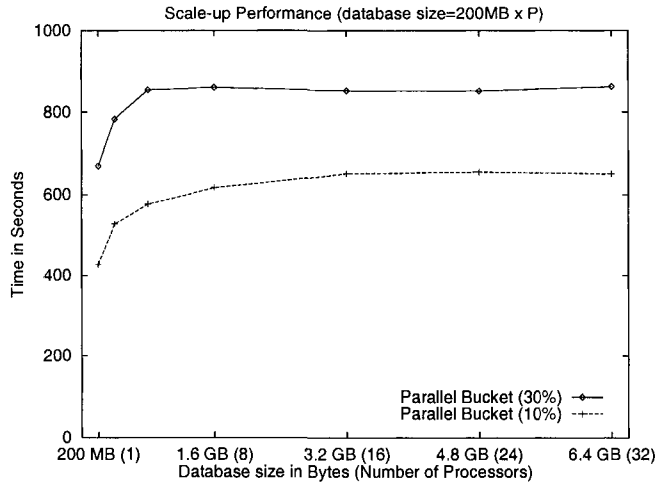
In this set of experiments we did not include comparisons with previous approaches for computing temporal aggregation in parallel [48, 130]. As we mentioned in Chapter 2, previous methods for parallel temporal aggregation are based on the aggregation tree. Therefore, they are limited by the amount of main memory available in the sys-

tem and cannot be used for large-scale aggregation.

For the scale-up performance measurements, we fixed the size of a database partition on each processor to 10 million tuples (*i.e.*, 200 MBytes), in a way that the entire database would grow proportionally as the number of processors increased. While the number of processors was varied from 1 to 32, the number of local buckets was fixed at 8. Thus, the total number of buckets used for data redistribution was $8 \times \mathcal{P}$, where $\mathcal{P}$ was the number of participating processors. The number of local buckets was determined from previous experiments (see Figure 3.12) based on the local partition size. Note that we used the sequential bucket algorithm for the case $\mathcal{P} = 1$.

In Figure 3.14(a), the scale-up plots were fairly close to a horizontal line, which indicated a nearly linear scale-up performance with respect to the increasing number of processors. This was corroborated by the fact that the time spent on data partitioning remained quite static when the number of processors was no less than eight. See Figure 3.14(b) for measurements (from the case of 10% long-lived tuples) separated into two processing stages. As the number of processors was increased from one to two, data partitioning time was increased by about 80 percent, due mainly to additional cost for message passing between processors. In contrast, the time spent on aggregation was increased only by 12 percent due to increased data replication. As the number of processors increased, however, the increase of overhead leveled off and became essentially flat above the four processor case, and thereby allowing nearly linear scale-up performance.

For the speed-up performance measurements, we fixed the size of the entire database to 320 million tuples (*i.e.*, 6.4 GBytes), and determined the size of a database partition based on the number of participating processors. That is, the size of a local partition on a single processor was 6.4 GBytes/$\mathcal{P}$. Due to a limited disk space on each processor, we started experiments from 8 processors and increased the number of processors up to 32, changing the size of local database partitions accordingly



(a) Scale-up Performance

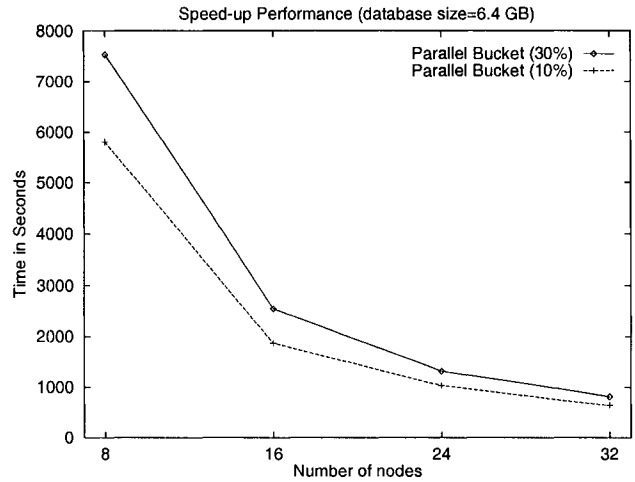
$\mathcal{P}$	**DB/$\mathcal{P}$** *MBytes*	**Partitioning** *Sec. (%)*	**Aggregation** *Sec. (%)*
1	200	71.0 (16.7)	355.9 (83.3)
2	200	127.8 (24.3)	399.3 (65.7)
4	200	176.3 (30.6)	399.2 (69.4)
8	200	207.2 (33.6)	408.7 (66.4)
16	200	196.2 (30.2)	454.2 (69.8)
24	200	183.5 (28.0)	470.9 (72.0)
32	200	185.4 (28.5)	464.7 (71.5)

(b) Separate Measurements (10% long-lived)

FIGURE 3.14. Scale-up Performance of Parallel Aggregation

from 800 MBytes to 200 MBytes. The resulting speed-up performance of the parallel bucket algorithm is shown in Figure 3.15(a).

It was surprising that a super-linear speed-up was observed when the number of processors increased from 8 to 16. From the separate measurements in Figure 3.15(b) (from the case of 10% long-lived tuples), such a super-linear speed-up can be attributed to the performance gain from local aggregation, which initially grew much faster than linear with the number of processors. That is, because the overall complexity of temporal aggregation is $\mathcal{O}(n \log n)$, processing two sets of $n/2$ tuples is faster than processing one set of n tuples. Note that the number of buckets used for



(a) Speed-up Performance

$\mathcal{P}$	**DB/$\mathcal{P}$** *MBytes*	**Partitioning** *Sec. (%)*	**Aggregation** *Sec. (%)*
8	800	801.0 (13.8)	4995.2 (86.2)
16	400	373.3 (20.0)	1497.3 (80.0)
24	300	248.4 (24.1)	782.2 (75.9)
32	200	185.4 (28.5)	464.7 (71.5)

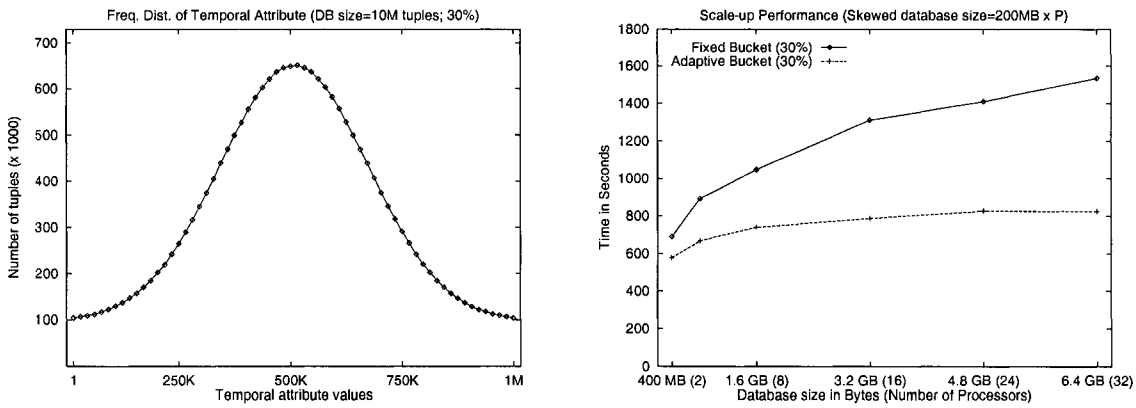
(b) Separate Measurements (10% long-lived)

FIGURE 3.15. Speed-up Performance of Parallel Aggregation

data redistribution increased proportionally to the number of processors. Thus, we conjecture that the overall aggregation cost was reduced by computing many smaller aggregations rather than computing a few larger aggregations.

3.4.5 Handling Data Skew

To evaluate how the parallel bucket algorithm deals with data skew, we generated synthetic data with Gaussian distribution of temporal attribute values. We assumed that equi-depth histograms were provided as a selectivity estimation mechanism. Figure 3.16(a) illustrates the distribution of start and end times of a local database on each participating processor. We compared the scale-up performance of the par-



(a) Data Distribution (30% Long-lived Tuples)

(b) Fixed-length vs. Adaptive Partitioning

FIGURE 3.16. Scale-up Performance for Skewed Databases. Measuring the Effect of Temporal Data Distribution on the Overall Performance

allel bucket algorithm with respect to two different data partitioning policies: (1) *fixed-length partitioning*, and (2) *adaptive partitioning*, as described in Section 3.3.1. In Figure 3.16(b), the adaptive partitioning significantly improved the performance for skewed data (with 30% long-lived tuples). Actually it took slightly less time than processing a database with uniform distribution using fixed-length partitioning. (Compare with Figure 3.14(a).) This results clearly demonstrate the effectiveness of the adaptive partitioning and scalable performance of the parallel bucket algorithm even at the presence of data skew.

3.5 Discussions for Further Extensions

In this section, we discuss application of the proposed temporal aggregation algorithms to queries with `GROUP BY` clauses. We also discuss further optimizations for evaluation of temporal aggregate queries on databases using coarse granularities and guidelines for selecting the number of buckets for the bucket algorithm.

3.5.1 Aggregate Queries with GROUP BY Clause

Aggregation queries are often combined with GROUP BY clauses. Thus, it is important to process an aggregation query with a GROUP BY clause efficiently. In traditional relational databases, two approaches based on *sorting* and *hashing* are commonly used. The *sorting* approach orders the tuples in an input relation by GROUP BY attribute. Aggregation is then performed by accessing the tuples in the sorted order. Aggregation for a group completes when a new grouping value is encountered. Then, aggregation for the next value will continue.

The *hashing* approach, on the other hand, requires building a hash table with entries of the form <*grouping value, aggregate value*>. As a relation is scanned, each tuple is fetched into its corresponding hash entry according to the value of the grouping attribute, and the *aggregate value* is updated. A limitation of this approach is that the number of groups should be small enough to fit in memory. If the number of groups outgrows the memory, a relation can be partitioned based on the grouping attribute values and each partition can be aggregated independently [26, 105, 113].

We can apply either of these two approaches to temporal aggregation. Once all the tuples in an input relation are sorted or hashed into buckets by their grouping attribute values, it is fairly straightforward to aggregate each group by using any of the temporal aggregation algorithms proposed in this paper. The choice of algorithm will be determined by the type of aggregation and the size of each group.

3.5.2 Fast Aggregation for Coarse Granularities

For temporal aggregate queries defined on *coarse* granularities, certain aggregate functions can be computed in linear time provided that there is enough memory available. We consider as coarse a granularity for which information of all its granules can be hold in main memory. For example, suppose that temporal attributes are defined on a time line of one million granules. Then, a COUNT aggregate can be computed in

linear time if the memory is large enough to hold two million counters. Two counters are needed for each granule to keep track of the number of tuples starting and ending at each granule. These counters are stored in a dense array so that each counter can be directly accessed by the timestamp value of a tuple.

The algorithm proceeds by scanning an input database updating counters for each tuple. Then, the array of counters is scanned to compute the `COUNT` aggregate function, in a similar way the balanced tree algorithm does. Note that it is not the database size but the number of distinct granules on a temporal domain that determines the memory requirements. We can apply this approach to `SUM` and `AVG` by allocating another word for each granule to store an extra aggregate value.

While this approach is very efficient for `COUNT`, `SUM` and `AVG` aggregate functions, it is not practical for `MAX` and `MIN` aggregate functions. The evaluation of `MAX` and `MIN` aggregate queries requires checking and updating aggregate values associated with all the granules overlapping the life span of a tuple. For example, consider a `MAX` aggregate. Let `maxaggr[]` denote an array that stores the max aggregate values. When a tuple `t` is fetched from a database, its value is compared with all aggregate values stored in the array segment `maxaggr[t.start:t.end]`, where `t.start` and `t.end` are the start and end timestamps of the tuple `t`. Specifically, for any `i` such that `t.start` $\leq i \leq$ `t.end`, if `maxaggr[i]` is smaller than the value of `t`, then `maxaggr[i]` is set to the value of `t`. This implies that the running time of this approach will be $\mathcal{O}(n \times m)$ in the worst case, where n is the number of tuples and m is the number of distinct granules in the database time line.

3.5.3 Optimal Number of Buckets

Knowing that the time complexity of the in-memory aggregation algorithms (*i.e.*, balanced tree and merge-sort) is $\mathcal{O}(n \log n)$, one can argue that the performance of the bucket algorithm can always be improved by increasing the number of buckets.

For example, the computational cost of an in-memory aggregation can be reduced from $\mathcal{O}(n \log n)$ down to $\mathcal{O}(n \log n - n \log \mathcal{N}_B)$, if $\mathcal{N}_B$ buckets are used. However, the more buckets are used for data partitioning, the shorter the time interval each bucket represents becomes. This implies that tuples are more likely to be replicated and the overall cost of aggregate computation and disk accesses will increase.

As yet, we have not found an analytic model or mechanism that can be used to obtain an optimal number of buckets. Instead, we have observed two factors that limit the number of buckets. These two factors are essentially characteristics of an input database, and can be used to help find a reasonable choice for the number of buckets if the database characteristics are known a priori.

The first factor is *data distribution*, which may affect the way a time line of an input database is partitioned and assigned to buckets. If tuples are evenly distributed over the time line, then the buckets will represent time intervals of an equal length and will store approximately the same number of tuples. On the other hand, if tuples are unevenly distributed, then the time intervals in the dense regions will be shorter than those in the sparse regions. These shortened time intervals will result in more tuples that are intersected by the boundaries of time intervals and replicated by the bucket algorithm, which will in turn increase the cost of aggregate computation and disk accesses. Thus, it is recommended to err on the side of a smaller number of buckets for an input database with skewed distribution.

The second factor is the *percentage of long-lived tuples.* Any tuple whose time interval is longer than that of a single bucket can be considered a long-lived tuple, because the tuple will necessarily be replicated by the bucket algorithm. The more long-lived tuples exist in the database, the more bucket algorithm costs for aggregate computation and disk accesses. Since large intervals of buckets will reduce the fraction of replicated (or long-lived) tuples, it is recommended to err on the side of a smaller number of buckets for an input database with a large population of long-lived tuples.

3.6 Summary of Results

We have developed new algorithms for computing temporal aggregates. The proposed algorithms provide significant benefits over the current state-of the-art in different ways. The balanced tree and merge-sort aggregation algorithms have improved the worst-case and average-case processing time significantly for small databases that fit in memory. We have also developed new sequential and parallel bucket algorithms based on novel data partitioning schemes. These algorithms can be used to compute temporal aggregates for databases that are substantially larger than the size of available memory, by processing data partitions in a sequential or parallel fashion. This represents an improvement over state-of-the-art techniques restricted by the amount of available memory. In particular, with the adaptive data partitioning scheme and the local and global meta arrays for partitioned data, we have demonstrated that the parallel bucket algorithm achieves scalable performance for large-scale databases by delivering nearly linear scale-up and speed-up, even at the presence of data skew.

Chapter 4
Survey: Finding Interesting Evolutions

Similarity search, or query by content, on time series data is used when applications need to compare the object's temporal evolution to a particular pattern known as the query pattern. It can be classified into *whole sequence matching* and *subsequence matching* [109]. In whole sequence matching, the query pattern is to be compared to data time series of the same length, whereas, in subsequence matching, we look for a consecutive subsequence within the data that best matches the query pattern.

In a similarity search query, the user specifies a query time series (pattern) $\overline{q}$ and a tolerance or degree of similarity for the entries in the answer set. The value of this tolerance can be explicitly or implicitly stated in the query. In an *ϵ-search query*, the user explicitly indicates ϵ, the tolerance value, whereas in a *k-Nearest Neighbor search query* (k-NN) the user merely indicates the number of entries that must be contained in the answer set. Hence, the tolerance value is implicitly indicated by the degree of similarity between the query pattern and the k entries in the answer. In any case, the solution to the search includes all entries in the database that satisfy the following condition.

$$Dist(\overline{x}, \overline{q}) \leq r,$$

where $Dist()$ is a domain specific distance function used to estimate the similarity between two time series, $\overline{x}$ is an entry in the data set of time series and r is the search radius. For an ϵ-search query, $r = \epsilon$. For a k-NN query, on the other hand, the value of r depends both on the value of k and on the data set. In particular, $r = Dist(\overline{x}_k, \overline{q})$, where, from all the entries in the data set, $\overline{x}_k$ is the k_{th} most similar entry to $\overline{q}$.

Similarity search on time series data has been extensively studied in the past

decade. The research community has followed two major trends in studying this problem. One line of research seeks after efficient mechanisms for processing similarity search queries, whereas the second investigates how to properly asses the similarity between two sequences. That is, it tries to find the best possible similarity model. These two approaches are complementary as one cannot look for similar patterns without a similarity model. Conversely, the best similarity model is of little use if we cannot efficiently process a query. In the rest of this chapter, we will study existing approaches for evaluating similarity search queries along these two lines of research.

4.1 Efficient Processing of Similarity Search Queries

To efficiently evaluate similarity search queries, it is important to organize the data time series in such a way that relevant time series can be retrieved without looking up the entire database exhaustively. It is easy to propose a simple technique for evaluating similarity search queries in which a time series data of length L is mapped into a point in L-dimensional space, and a spatial access method such as the R-tree [8, 56] is used to index them. However, it is not uncommon that time series data are sampled during a relatively long period of time. Thus, a direct application of spatial access methods will require mapping time series data into a very high dimensional vector space. In consequence, this approach would likely suffer performance degradation due to reduced pruning power of an index, a phenomenon known as the *curse of dimensionality.*

4.1.1 The Curse of Dimensionality

The phrase "curse of dimensionality" was coined by Richard Bellman [9] to indicate that sampling is not an appropriate strategy for the optimization of multi-variate functions. If this technique is used, the number of evaluations required to optimize the function grows exponentially with the number of dimensions. In the database

field, we face the curse of dimensionality when trying to organize high-dimensional data for efficient searches.

Conventional querying mechanisms for 1-dimensional data use indexes to speed-up searches. These indexes take advantage of a linear ordering among data objects to organize and search data. Unfortunately, high-dimensional data lack a linear ordering that could be exploited by an index. Access methods must be adapted to work on a spatial ordering instead. However, high-dimensional spaces have peculiar properties that interfere with our quest for developing efficient access methods. In particular, the behavior of data and distance distribution in high-dimensional spaces have a significant impact on the efficiency of indexes. We explain these peculiarities next.

First, data is sparsely distributed in high-dimensional spaces [12]. While the space grows exponentially with the number of dimensions, the number of entries in the database does not. Let us illustrate the sparseness of data in high-dimensional spaces with the following example. Consider, without loss of generality, a uniformly distributed data set within the unit hypercube. For this data set, we can calculate the probability of finding a point in a rectangular range subspace as

$$P(s) = s^d,$$

where s is the side of the hypercube defining the subspace. If we define a cubic subspace of side $s = 0.9$, we would expect to find 81% of the data in a 2-dimensional space. However, if the space is 50-dimensional, the subspace will only contain 0.51% of the data.

Second, point to point distance distribution is concentrated in a histogram with a small variance [27]. In other words, the difference between the minimum and maximum distances to a given point is not significant in high-dimensional space. This property was captured by Beyer *et al.* [17] in the following theorem.

Theorem 1 (Beyer's Theorem). *Let D_{min} be the distance from a point in d-dimensional space to its nearest neighbor and let D_{max} be the distance to its farthest neighbor. If*

the point to point distance distribution follows a curve with a small variance, as we increase the dimensionality of the space, it happens that the difference between the nearest and farthest distance does not increase as fast as the nearest distance. That is, If $lim_{d\to\infty} var(\frac{d(x-q)}{E[d(x-q)]}) = 0$, *then*

$$\frac{D_{max_d} - D_{min_d}}{D_{min_d}} \to 0.$$

When evaluating a similarity search query, search algorithms traverse the index structure and prune irrelevant subsets of data (*i.e.*, branches of the index) based on distances from a set of candidates. Because there is no significant difference in the magnitude of these candidate distances, Theorem 1 indicates that using an index to selectively retrieve candidates will necessarily degrade into an exhaustive search [3].

4.1.2 The GEMINI Paradigm

To address the problem of indexing large multimedia objects, for which indexing is either not possible or not efficient, Faloutsos proposed the *GEneral Multimedia INdexIng* (GEMINI) paradigm [42]. The idea is to apply a transformation that produces an indexable representation of the multimedia object. This representation is usually a vector that conveys relevant information about the main characteristics of the object and it is usually known as *feature* or *signature vector*. Feature extraction from time series data has been known as *dimensionality reduction* because it generally represents a time series with a small number of values. However, some techniques for extracting features from time series do not necessarily reduce the total number of values but only their precision (*e.g.*, quantization). During the process of extracting features, some information about the time series data is lost. That is, if we reconstruct a time series object from its representation, we can only obtain an approximation of the original object. The difference between the original and the reconstructed objects is known as the *approximation error*. Because they are more accurate, for the rest of

this dissertation, the terms feature extraction and *lossy compression* are favored over the term dimensionality reduction of time series data.

4.1.3 Extracting Features from Time Series Data

Many promising techniques have been proposed to generate a small representation of a time series data for improving similarity search performance. In this section, we present a review of these techniques, as well as the context on which they have been presented.

Discrete Fourier Transformation (DFT): On their seminal work on indexing time series data for similarity search, Agrawal *et al.* [4] proposed the use of Discrete Fourier Transformations (DFT) to represent time series data as vectors in a relatively low dimensional frequency space. This proposal is based on Parseval's theorem, which indicates that time series data can be transformed from the time domain to the frequency domain while preserving its energy. One important implication of this theorem is that the Euclidean distance between two time series is also preserved under this transformation. These feature vectors are then indexed using a spatial access method such as the R*-tree [8]. The proposed approach is called F-index. The authors suggested that, because the values in a signal are not random but rather they follow some pattern, most of real signals need only a few DFT coefficients for a close approximation. In the specific type of similarity search they studied, whole sequence match, the Discrete Fourier Transform is also applied to the query pattern. The search is conducted using the index and the first f_c DFT coefficients of the query pattern, where f_c is the same number of DFT coefficients used to build the index. Searching the feature index produces a superset, usually known as *candidate set* of the solution to the query. In the last step of the query processing, *false alarms* are filtered out of the candidate set to produce the final query result.

The work by Agrawal *et al.* was later generalized by Faloutsos *et al.* [44] to ac-

count for subsequence match queries. Faloutsos *et al.* presented an approach in which all subsequences of length w in a data time series are extracted using a sliding window. Each of these subsequences is then transformed using DFT, keeping the first f_c coefficients for indexing purposes. Because a data time series of length L, generates $L-w+1$ subsequences of length w, the idea of using a naïve approach that inserts individual subsequences in an index structure is inadequate. In studying this problem, Faloutsos *et al.* observed that the DFT of adjacent subsequences generates trails in the f_c-dimensional space. Instead of indexing individual subsequences (actually, their DFT), they proposed to group adjacent subsequences together and insert only the their Minimum Bounding Rectangle (MBR) into an index. They also indicated that the expected cost of a search query is a function of the size of the MBRs in the index (*i.e.*, a larger MBR is more likely to be "hit" by the query). Therefore, finding the best grouping of subsequences is an optimization problem, where the expected cost of a search query is minimized. Note that this problem is restricted to grouping only adjacent subsequences, corresponding to sub-trails in the f_c-dimensional space. This restriction gives name to the proposed approach, the Sub-Trail index (ST-index).

Once the ST-index has been built, there are two approaches for searching it. *Prefix* search uses only the first w values of the query pattern while searching the index. *MultiPiece* search, on the other hand, uses all disjoint subsequences of length w generated from the query pattern during the search process. Both of these approaches will produce a candidate set with false alarms that need to be filtered out. For query sequences of length $L < w$, however, the query processor would have to resort to linear scan on raw data.

Rafiei and Mendelzon [102] have also used the discrete Fourier transform for the retrieval of similar time series. In their work, they proposed an improvement over the DFT-based indexing techniques for time series. In particular, they indicated that the last f_c Fourier coefficients can be used for distance computation without storing them in the index. They noted that, because the DFT of a real-valued time series

is symmetric with its center, every coefficient at the end is the complex conjugate of a coefficient at the beginning and as a strong as its counterpart. Using this properties, combined with standard indexing techniques such as the F-index proposed by Agrawal *et al.* [4], they observed in their experiments performance improvements of more than a factor of two with respect to previous techniques that used DFT.

Discrete Wavelet Transformation: Discrete Wavelet Transformations (DWT) are sometimes used as alternatives to DFT for signal and image processing. The point of difference between these two transformations is that DFT maps a time series into a frequency domain while DWT maps it into a representation that allows localization both in time and frequency domains [21, 96, 127]. The use of wavelet transformations for similarity search on time series data was first hinted by Agrawal *et al.* [4], when they suggested that any *orthonormal* transformation could be coupled to the concept of F-index. The use of the Haar wavelet transformation (*i.e.*, an orthonormal transformation) for processing similarity search on time series data was proposed by Chan and Fu [21]. They proposed replacing the DFT for the Haar wavelet transform in the F-index. The empirical evaluation they conducted indicated performance gains in the use of Haar with respect to DFT for similarity search in time series.

Popivanov and Miller [96, 97] proved that not only orthonormal transformations (such as DFT and Haar), but also *bi-orthonormal* wavelets can be combined with the F-index for efficiently processing similarity search queries on time series data. In particular, they proved that, while the Euclidean distance is not generally preserved under bi-orthonormal wavelets, it can be bounded to guarantee the triangular inequality required by the spatial access method used by the F-index. In their experimental evaluation, they also reported performance improvements over DFT. Despite these results by Popivanov and Miller and Chan and Fu, the dominance of DWT over DFT for efficient similarity search on time series data is still under debate. In fact, recent studies by Keogh and Kasetty [71] and Wu *et al.* [127] have shown that the observed

performance gains by using DWT over DFT (or vice versa) seem to depend on the data sets used for the empirical evaluation (a phenomenon known as *data bias* [71]). Therefore, it is unclear whether one kind of transformation is better than the other.

Singular Value Decomposition: This method is also known as the *Karhunen-Loeve Transform* (KLT) and it is often cited in the database literature as *Principal Component Analysis* (PCA), a term also used in the statistical literature [22]. PCA is used when we suspect that the variation of a multi-variate data set can be explained with few linear combinations of the original variables. In such case, although p components are required to reproduce the total system variability, much of this variability can be accounted for by a small number, k, of the principal components. That is, there is almost as much information in the k components as there is in the original p variables [67]. In database research, we are often interested in achieving data compression by replacing the initial p variables with the k principal components. This reduces the data, from one dataset consisting of n entries with p variables to one consisting of n entries with k principal components, while preserving most of the information conveyed by the original data set.

Korn *et al.* [82] developed enhancements to the singular value decomposition technique for processing queries on large data sets of time series. The intuition behind this approach is that, while some entries may be approximated poorly by SVD, extra information can be used to establish bounds on the approximation error of individual entries. Korn *et al.* proposed to store the k principal components resulting from SVD and complement them with the reconstruction errors (deltas) of some entries (those with the highest errors). This approach, called *Singular Value Decomposition with Deltas* (SVDD), can evaluate queries using the approximation of the data time series defined in the resulting k-dimensional space. While it is possible to know what the reconstruction error is for outliers, this information is not useful for similarity search queries.

FastMap: This technique was developed by Faloutsos and Lin [43] for mapping objects embedded in an unknown L-dimensional space into points in k-dimensional space. The process starts with a matrix of $n \times n$ distances between objects and finds the k-dimensional space where these distances are preserved. For time series data, Yi *et al.* [132] proposed the use FastMap for similarity search. They proposed reducing dimensionality of time series data using FastMap and then indexing the resulting points in k-dimensional space using a spatial access method as in the F-index.

Piecewiese Approximation: Another approach for reducing dimensionality of time series data is to decrease the resolution of time series. Shatkay and Zdonik [112] suggested breaking a time series data into meaningful subsequences and storing approximate and compact representations of the subsequences as mathematical functions. In particular, they used linear functions to represent the stored sequences. There is a user-defined parameter, θ, that characterizes the slope of these linear functions as positive ($> \theta$), negative ($< \theta$), or neutral. Similarity queries can be specified as regular expressions on an alphabet of positive, negative, and neutral slopes.

Keogh and Smyth [73] suggested a piecewise linear representation for time series data. They decided on a bottom-up approach for finding the k segments that best represent a time series. Under this approach, a pair of adjacent segments are merged at each step. The pair of segments to be merged is selected based on the least increase of squared error of the approximation. The merging process stops when the amount of error introduced when going from k to $k-1$ segments increases significantly. Because the resulting number of segments might be different for different time series, search algorithms must exhaustively explore all data representations in the database.

Morinaka *et al.* [91] proposed to represent time series for sequence matching queries using sequences of linear segments. In the proposed method, a time series is transformed into a bounded sequence of approximated linear segments called Δ-SEALS. One linear segment is the longest possible linear segment whose accumulated

error does not exceed a given "deviation bound" (*i.e.*, Δ). The algorithm proceeds by incrementally approximating a subsequence by a linear segment until sum of squared errors of the line segment to the actual data is greater than Δ. At this point, a new line segment is created and the algorithms repeats these steps until all values in the time series have been used. After segmenting the query sequence, the search algorithm scan the linear segments of the query and data sequences along the time dimension. That is, it compares the data to the query at every possible offset to estimate their similarity.

Keogh and Pazzani [75] also used piecewise linear segmentation to represent a time series as a k-bit string, where k is the number of segments used for the transformation. Under this approach, each segment of the time series is discretized by a bit value indicating whether the values in the segment are mostly rising or not. The bit pattern obtained this way is called *Shape To Bit vector* (STB) and yields a value indicating a bin where the pointer to the time series is stored (*i.e.*, a kind of hashing). On every bin, the distance matrix of all pairs of sequences represented in the bin is computed. This matrix defines a metric space within the bin that helps pruning false alarms during similarity search. In the search process, the query pattern is also represented as an STB and compared to all the bins defined in the index. If the distance from the query STB to a bin is larger than the current best result, the bin is simple ignored.

Yi and Faloutsos [131], and Keogh *et al.* [70] independently suggested the representation of time series data with the aggregate value of their segments. Under this approach, called Segmented Means (SM) [131] or Piecewise Aggregate Approximation (PAA) [70], a time series data is partitioned into k segments of equal length. Each of these segments is represented by their aggregate value. In particular the `AVG` aggregate function is used to represent a segment with a single value. After this transformation has been applied, a data time series is represented by a sequence of k values (*i.e.*, the mean value of each segment). This feature vector can be considered as a point in k-dimensional space and indexed using any spatial access method as

in the F-index. For a similarity query, the query sequence is also partitioned into k segments of equal length and the resulting vector used to search the index.

More recently, Keogh *et al.* [74] proposed a new approach, called Adaptive Piecewise Constant Approximation (APCA), which uses variable-length segments in an attempt to better approximate time series data. The APCA representation x', of a time series data $\overline{x}$, is a sequence of $2k$ values of the form $x' = \{<v_1, r_1>, ..., <v_k, r_k>\}, r_0 = 0$, where k is the number of segments, v_i is the mean value of the data points in the i_{th} segment and r_i is the right end point of the i_{th} segment. This new representation is a generalization of the Segmented Means approach allowing segments of different length to minimize the approximation error.

The F-index approach cannot be directly used with APCA because distances defined on this representation do not satisfy the triangular inequality. To use an index, APCA representations must be grouped in a special kind of bounding rectangle. An MBR R, containing a group of APCA representations with k segments, defines k rectangular regions in a 2-dimensional *time-value* space, $R = \{R_1, R_2, \ldots, R_k\}$. R_i is the minimum bounding rectangle (in the *time-value space*) containing the i_{th} segments of all the time series in the group. The distance function used by the index to organize data is computed with respect to the region formed by the union of these 2-dimensional rectangles.

Quantization: While not specifically developed for time series, quantization techniques can be applied to approximate time series data. One of such techniques is the *Vector Approximation File* (VA-file) proposed by Weber *et al.* [126]. The VA-File divides a d-dimensional data space into $d \times 2^b$ rectangular cells where b denotes a user specified number of bits. It allocates a unique bit-string of length $d \times b$ for each cell. Data points that fall into a cell are then approximated by the corresponding bit string. Instead of hierarchically organizing these cells in an index structure, the VA-File itself is simply an array of these compact approximations [125, 126]. A

variation of the VA-file, the VA$^+$-file, was proposed by Ferhatosmanoglu *et al.* [45] to efficiently process similarity search queries on non-uniform high dimensional data sets. The VA$^+$-file requires applying the Karhunen-Loeve Transform (KLT) to the data set before computing the quantization on the transformed data.

Also using vector approximations, Sakurai *et al.* suggested the *Approximation Tree* (A-Tree) [108]. A Key concept of the A-Tree is the introduction of *Virtual Bounding Rectangles* (VBR) to approximate Minimum Bounding Rectangles (MBR) and data objects in an index structure. A VBR is a quantized representation of a rectangle or a point relative to the MBR of the parent node in the index. Compared to the VA-File, where the data objects are approximated with respect to a *global reference space*, the use of *relative approximations* (approximations relative to a local reference space) in the A-Tree provides more accurate representations of the data objects. The idea of combining a hierarchical space partitioning and local quantization was also explored by Berchtold *et al.* [11] in their IQ-tree. In fact, these two approaches were proposed almost simultaneously and present the same general idea. For the rest of this work, we only use the A-Tree as a representative of this approach that combines an index structure with quantization.

The VQ-Index presented by Tuncel [124], is another quantization technique for similarity search in multimedia databases. However, the VQ-index is used only for approximate nearest neighbor search. Since we are interested in exact queries, we will not discuss further details about this technique.

4.2 Estimating Similarity

When doing search by content in a time series database, we need a model to measure the similarity between two time series. These models are usually designed to be robust to transformations such as translation or scaling depending on the application domain [29, 73]. In this section, we present a review of different models that have

been proposed for estimating similarity in time series data.

Euclidean distance: Perhaps the most widely used model of similarity has been the Euclidean distance. Euclidean distance has been used with almost every available dimensionality reduction technique. Among the methods presented in Section 4.1.3, Discrete Fourier Transformation [4, 44, 102, 100], Discrete Wavelet Transformation [21, 22, 68, 96, 97, 120], Piecewise Aggregate Approximation [70] (Segmented Means [131]), Adaptive Piecewise Constant Approximation [74], Shape To Bit [75], FastMap [43], Singular Value Decomposition [45, 82], and Quantization [11, 45, 108, 124, 126] make use of Euclidean distance for measuring similarity. However, it has been noted by the research community that Euclidean distance has several drawbacks assessing the similarity between two time series. In particular, because it considers absolute values, Euclidean distance is sensitive to outliers. An outlier is a measurement that does not reflects the actual state of the studied phenomenon. In addition, it can only be used when comparing two sequences of the same length (or equi-length subsequences of them) [5, 51, 101, 100].

L_p norms: The infinity norm L_∞ was used by Agrawal *et al.* [5] as part of a more complex similarity model. They proposed a model that considers that two time series are similar if they have similar subsequences. To do this, subsequences of length w, called windows, are extracted from the time series being compared and normalized to compensate for amplitude scaling and offset translation. Windows from the query and data time series are matched and considered similar if their L_∞ distance is less than a given threshold. These windows are latter "stitched" to form pairs of long similar non-overlapping subsequences. Two sequences are be similar if they have enough non-overlapping time ordered pairs of sequences that are similar.

L_p norms, in general, are used by Yi and Faloutsos [131] as their similarity model for whole and subsequence match using the segmented means (SM) data representation. While Keogh *et al.* [70] presented their PAA data representation using the

Euclidean distance to measure similarity, any L_p norm can be used with it. Other techniques that can use an arbitrary L_p norm as a metric of similarity are APCA [74] and Quantization [11, 45, 108, 124, 126].

Dynamic Time Warping (DTW): There are some cases where Euclidean distance, and L_p norms in general, may not be entirely adequate for estimating similarity. The reason is that L_p norms are sensitive to distortions in the time axis. To avoid this problem, similarity models should allow some elastic shifting of the time dimension to detect similar shapes that are locally out of phase [29]. This is the case of Dynamic Time Warping (DTW), introduced to the context of time series by Berndt and Clifford [13]. DTW is used when the size of two time series is different or when it is required to match sequences that are out of phase [13, 69, 72, 77, 93, 132]. One attempt to match sequences out of phase was presented by Rafiei and Mendelzon [101] when they proposed a model that included transformations for handling global time scaling. Unfortunately, global time scaling does not compensate for local variations of the time axis and it is still sensitive to them.

One significant problem of DTW is its high computational cost, comparing two time series of length n and m takes $\mathcal{O}(n \times m)$ time. To reduce this cost, Yi *et al.* [132] proposed the use of FastMap in combination with a lower-bounding distance function to DTW based on the minimum and maximum values of a sequence. FastMap is used to reduce the dimensionality of the objects being compared and, in consequence, the computational cost of DTW. The lower-bounding distance is used to prune away non-qualifying elements. However, because the Dynamic Time Warping distance does not satisfy the triangle inequality [132],the use of FastMap might result in a number of false dismissals.

The problem of the high cost of DTW was also addressed by Keogh *et al.* [72]. They introduced the Piecewise Dynamic Time Warping (PDTW) distance, which approximates the dynamic time warping distance using only the PAA [70] data represen-

tation of the time series being compared. Because it only provides an approximation to the actual DTW distance, this approach introduces false dismissals. In addition, it is sensitive to the arguments provided by the user regarding the compression ratio of the data representation. That is, the number of resulting false dismissals is sensitive to the number of segments used for the PAA representation (PAA is a lossy compression technique, the approximation error increases as we reduce the number of segments). A similar approach was presented by Chan *et al.* [22] using the Haar Wavelet Transformation as their data transformation. The model they proposed is called Low Resolution Time Warping and it provides an approximation of the DTW between the two sequences being compared. Because it only approximates DTW, this model cannot guarantee no false dismissals either.

To make PDTW more robust to false dismissals, Chu *et al.* [29] later improved on the PDTW similarity by developing probabilistic models for the approximation error. They proposed the Iterative Deepening of Dynamic Time Warping (IDDTW) similarity model based on the idea of generating data approximations using different compression ratios. When evaluating a similarity search query, new candidates are iteratively compared using the different compression ratios of the PAA data representation. The highest compression ratio is used first to obtain a rough approximation of the DTW distance. If, according to the error model, this distance is out of the tolerance given by the user, the candidate is pruned away. Otherwise, the candidate is checked using the PDTW defined at the next lower level of compression. This process stop when the candidate is pruned away or when the actual DTW distance is computed.

Park *et al.* [93] proposed an indexing technique for similarity search on time series data based on time warping distance. Their approach is based on categorizing the values found in a time series to create a string. The strings so obtained are inserted into a disk-based suffix tree for indexing and searching. Keogh [69] observed that the suffix-tree used by Park *et al.* can be orders of magnitude larger than the original data

and proposed an indexing technique based on a restricted time warping distance. He restricted the range of warping for the distance and approximates a time series data by a bounding region. He approximates this bounding region by the PAA approach and provides a distance function to this region that lower-bounds the restricted time warping distance. This idea was recently improved by Zhu and Shasha [137]. They proved that the approximation to the envelope does not need to contain every point in the envelope (as Keogh [69] did) but only every approximated point in the envelope. This property is termed *container-invariant.* A container-invariant transformation of an envelope guarantees no false dismissals for dynamic time warping queries [137].

Other models of similarity Perng *et al.* [94] presented the *landmarks* a model of similarity for time series data. This model is based on the fact that people recognize patterns in charts by identifying important points. A landmark in a time series is a point of importance. Mathematically, landmarks are those points whose derivative equals zero. Two sequences are similar if their landmark representations can be transformed and be within a given threshold of each other (in time and value dimensions).

Keogh and Smyth [73] presented a probabilistic model on which linear segments representing a time series can be deformed on both the time and value dimensions. These linear segments are extracted from the time series by finding local features (*e.g.*, local minima and maxima). Two time series to be compared are aligned with respect to the starting point of each of their segments and the similarity between them is given by the minimum standard deviation between the values of the segments.

Chapter 5
An Index-based Approach for Evaluating Whole Sequence Match Queries

In this chapter, we focus on the problem of whole sequence match and propose a simple and elegant paradigm for indexing time series data. While previous approaches organize time series using their feature vectors, the proposed approach, called *Skyline Index*, adopts new *Skyline Bounding Regions (SBR)* to group and organize time series data using their native *time-value* space. Because of this, the skyline bounding regions allow us to define a distance function that tightly lower-bounds the distance between a query object and a group of time series data, which translates into better index performance. We summarize the benefits of building an index based on the skyline bounding regions in the following list.

- *Reducing index access.* Tight lower-bounding distances can be defined based on the SBR. This reduces the number of index pages accessed during search.

- *Reducing data access.* A reduction on the number of data objects fetched can result from the combined effect of the lower-bounding distance to an SBR and the lower-bounding distance to each of the entries in the SBR. We provide theoretical proof and empirical evidence of this effect in Sections 5.2.7 and 5.3.4, respectively.

- *Orthogonal to data approximations.* Different approximations can be used to represent data entries and skyline bounding regions. For instance, DFT, DWT, PAA, or APCA approximations can be used to represent data entries in a leaf

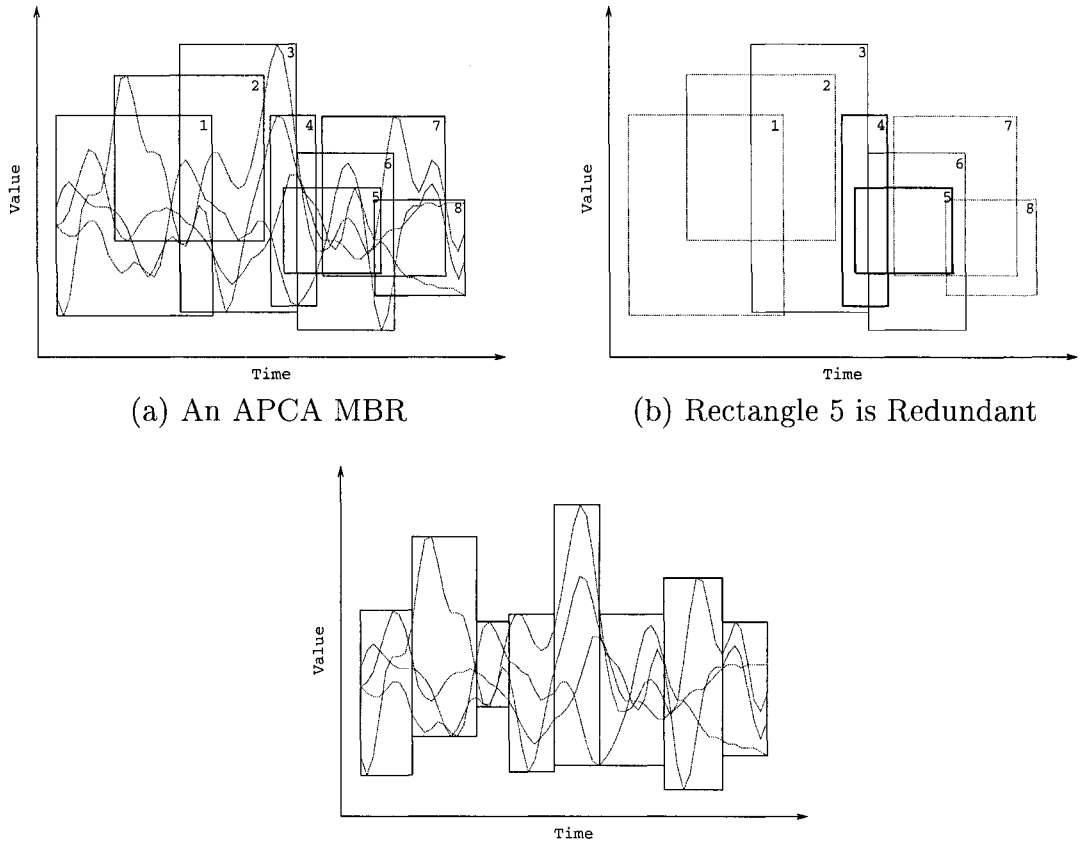
Notation	Description
O	universe of time series objects
D, $D \subset O$	a data set of time series
N, $N = \|D\|$	number of entries in data set
L, $L \in \mathcal{N}$	length of each time series object
$\overline{x}$, $\overline{x} \in D$	time series data object of length L
$\overline{q}$, $\overline{q} \in O$	time series query object of length L
$\mathcal{F}$	set of feature vectors
$x' \in \mathcal{F}$	the feature vector of the time series $\overline{x}$
R	a set of bounding regions
$d_{feature} : O \times \mathcal{F} \rightarrow \mathcal{R}^+$	distance from the query to a feature vector
$D_{Region} : O \times R \rightarrow \mathcal{R}^+$	distance from the query to a bounding region
$d_{Apca} : O \times \mathcal{F} \rightarrow \mathcal{R}^+$	distance from the query to an APCA feature vector
$D_{Apca} : O \times R \rightarrow \mathcal{R}^+$	distance from the query to an MBR in the APCA index
$D_{Sphere} : O \times R \rightarrow \mathcal{R}^+$	distance from the query to a sphere in the M-tree
$LB_{Keogh} : O \times R \rightarrow \mathcal{R}^+$	distance from the query to an SBR in the Skyline index

TABLE 5.1. Symbolic Notation Used in this Chapter

node. In internal nodes, skyline bounding regions can be approximated either by PAA or APCA.

The rest of this chapter is organized as follows. In Section 5.1, we start by stating the limitations of APCA, the state-of-the-art approach for processing whole sequence match queries described in Section 4.1.3. After this, the Skyline Index approach is presented in Section 5.2. The results of experimental evaluation are given in Section 5.3. In Section 5.4, we present some discussions of the Skyline Index with respect to previous work. Finally, Section 5.5 summarizes the contribution of this chapter.

For the notation used in this chapter, the reader is referred to Table 5.1.



(a) An APCA MBR

(b) Rectangle 5 is Redundant

(c) An MBR without Overlap

FIGURE 5.1. Limitations of the APCA Bounding Regions. Redundant Information is Stored in the Index

5.1 Limitations of APCA

While the APCA representation is the most promising technique for efficient similarity search of time series data, there is a potential weakness in the method used to define bounding regions during indexing. In particular, the 2-dimensional rectangles, $R_1, R_2, \ldots, R_M$, defined by the MBR of an index node, can overlap. Overlapping rectangles could have negative effect on the search performance. This effect can be explained in two different ways. Figure 5.1(a) shows a group of time series data and a set of 2-dimensional rectangles defined by an APCA MBR, R. For this example, we

have used 8 segments in the APCA representation, and the APCA MBR defines 8 2-dimensional rectangles. In Figure 5.1(b), we have eliminated the time series data for better visualization of the 2-dimensional rectangles. In this figure, it becomes clear that rectangle 5 is redundant because it is completely covered by rectangles 3 and 6. Therefore, the same shape for R can be defined using less space in the index node (*i.e.*, a smaller key). In this example, we could have reduced the key size by 12.5%, had we decided to ignore rectangle 5. For this reason, after eliminating redundant rectangles, it would be possible to reduce the index size and consequently improve search performance.

Alternatively, using the same amount of space in the index node, a better MBR could be defined if we used a representation free of overlaps. That is, an MBR can represent more accurately the collective shape of a group of time series data. This case is illustrated in Figure 5.1(c), where we have used the same number of rectangles as in Figure 5.1(a), except that care has been taken to avoid overlaps. It is clear that the amount of *dead space* (*i.e.*, the portion of the rectangles not containing data) in Figure 5.1(c) is less than the dead space in Figure 5.1(a). Therefore, tighter lower-bound distances to a group of time series can be defined resulting in an improved performance for similarity search. Note that we are referring to the overlap among the 2-dimensional rectangles, $R_1, R_2, \ldots, R_M$, defined by a $2M$-dimensional APCA MBR R. Let us call this type of overlap *internal overlap*, which is different from the overlap between MBRs (*i.e.*, *external overlap*) commonly observed in spatial access methods.

We conducted a preliminary experiment to verify our intuition on the existence of *internal overlap* in the MBRs defined by the APCA approximation. To quantify the amount of *internal overlap* present in APCA indexes, we define the following indicator,

$$O(R) = \frac{\sum_{i=1}^{M} Area(R_i)}{Area(\cup_{i=1}^{M}(R_i))},$$

Segments (M)	$\Sigma Area(R_i)$	$Area(\cup(R_i))$	$O(R)$
8	1.180	0.515	2.291
16	3.140	0.867	3.622
32	8.679	1.564	5.549

TABLE 5.2. Internal Overlap of the Bounding Regions Defined by APCA

which is the ratio of the sum of the areas of all the 2-dimensional rectangles defined by an APCA MBR R to the area of R. A large $O(R)$ value indicates that the amount of *internal overlap* is considerable and there is room for improvement in the search performance by using an internal overlap-free bounding region.

For this preliminary experiment, we built 3 different indexes using the APCA representation with 8, 16, and 32 segments, respectively (*i.e.*, M = 8, 16, or 32). Because each segment requires two values, the dimensionality of the approximations was 16, 32, and 64, respectively. We computed $O(R)$ for all leaf nodes in each index and present the average of our measurements in Table 5.2. This table shows that the sum of the areas of all rectangles was up to 5 times the area of the MBR, indicating a considerable amount of *internal overlap.* We should note that we have purposely ignored the units of the values in columns 2 and 3 of Table 5.2. The units for both columns are defined in the $time \times value$ domain. However, since we are only interested in the ratio between the two values, their units are irrelevant for our study. These results motivate our approach that groups time series data by new *Skyline Bounding Regions (SBR)* to provide tighter lower-bounding distances.

In all our preliminary experiments presented in this chapter, we used a data set composed of 99,000 electroencephalograms (EEG) of length 256. This data arose from a large study to examine electroencephalogram correlations of genetic predisposition to alcoholism. It contains measurements from electrodes placed on subject's scalps which were sampled at 256 Hz for 1 second. We obtained this data set from the UCI KDD data archive [61].

5.1.1 The k-Nearest Neighbor Search Algorithm

Similarity search and, in particular, k-nearest neighbor (k-NN) search can be implemented using the APCA approximation for time series data. Keogh *et al.* used a variation of Seidel's optimal multi-step k-NN search algorithm [110] and showed that the APCA approximation can be combined with an index for answering similarity search queries [74].

We include Keogh's k-NN search algorithm here (Algorithm 4) since we need it to explain some of the concepts presented in this chapter. Algorithm 4 works on multidimensional index structures such as the R-tree. The leaf nodes contain approximate representations (feature vectors) of time series data. These feature vectors are used to calculate the lower-bounding distance $d_{feature}()$ between a query and a data time series. Each entry in the internal nodes is a minimum bounding region (MBR). An MBR can be a bounding rectangle of the feature vectors of its sub-tree. It can also be a region defined on the approximations of the actual data contained by the MBR. This internal entry is used to calculate the lower-bounding distance $D_{Region}(q, R)$ between a query q and an MBR R. Note that $d_{feature}()$ and $D_{Region}()$ are only generic function names. They are defined depending on the dimensionality reduction techniques used for both data and bounding regions.

The correctness of this k-NN search algorithm (Algorithm 4) with respect to the lower-bounding distances provided by the index is summarized by the following lemma.

Lemma 1. *Given a query time series $\overline{q}$, and the MBR R of an index node, the answer for a k-NN search query using Algorithm 4 is correct if and only if the following two conditions on $d_{feature}()$ and $D_{Region}()$ are satisfied.*

1. $d_{feature}(\overline{q}, x') \leq Dist(\overline{q}, \overline{x})$, $\forall\ \overline{x}$ *in the database. This condition is also known as the* ***contractive property****.*

Algorithm 4: K Nearest Neighbor Search

```
Input: (q, k)
// q̄ is the query object.
// k is the number of nearest neighbors searching for.
1  enqueue(prio_queue, root, 0);
2  while prio_queue is not empty do
3      entry e ← dequeue(prio_queue);
4      switch the object pointed by e do
5          case a data object
6              result ← result ∪ {e};
7              if |result| = k then return result;
8          case an entry in a leaf node
9              fetch the actual data object x̄ pointed by e;
10             enqueue(prio_queue, x̄, ||q̄ − x̄||_2);
11         case an index leaf node
12             for each data approximation x' in leaf node e do
13                 enqueue(prio_queue, x', d_feature(q̄, x'));
               end
14         case an index internal node
15             for each child node R in e do
16                 enqueue(prio_queue, R, D_Region(q̄, R));
               end
       end
   end
```

2. $D_{Region}(\overline{q}, R) \leq Dist(\overline{q}, \overline{x})$, $\forall\ \overline{x}$ *in the MBR* R. *This condition is referred to as the* ***group lower-bound property.***

In Lemma 1, $Dist(\overline{q}, \overline{x})$ is the distance function used to estimate the similarity between two time series. For the rest of this chapter, we assume $Dist(\overline{q}, \overline{x}) = \|\overline{q} - \overline{x}\|_2$.

Lemma 1 gives us the guidelines for designing index organizations and dimensionality reduction techniques. The first property is usually supported by feature extraction or dimensionality reduction techniques. The second property should be supported by the indexing technique of choice. Note that, in Lemma 1, there is no mention of the *triangular inequality* about the feature distance $d_{feature}()$. As long as the two conditions are satisfied, Algorithm 4 will yield correct results.

For brevity, we omit the proof of Lemma 1. The interested reader can find a detailed proof of this lemma in the original APCA work [74].

5.2 Skyline Index Organization

In this section, we describe the *Skyline index*. First, we introduce the *Skyline Bounding Region (SBR)*, which is the core idea of the proposed approach. Then, we describe the fundamental details regarding a lower-bounding distance function for SBRs, and the adoption of SBRs for indexing and searching time series data.

5.2.1 Skyline Bounding Regions

Under traditional methods for dimensionality reduction, time series data are first mapped into points in a low-dimensional feature space, and the points (*i.e.*, feature vectors) are then grouped into MBRs to be stored in a hierarchical index structure. In this section, we present an alternative way to organize time series data, which enables dimensionality reduction techniques to maintain high fidelity of approximation, and thereby improving the performance of similarity query processing.

A time series data exists in a two-dimensional *time-value* space. If a minimum bounding rectangle (MBR) defined in this space is used to bound a group of time series data, the external overlap between MBRs will be large, and the search performance will suffer from this. In order to minimize the overlap between MBRs, it is desirable to approximate a group of time series data with tighter bounding regions. To achieve this goal, we propose to use *skylines* to bound a group of time series data. This bounding region, called *Skyline Bounding Region (SBR)*, is defined as follows.

Definition 5 (Skyline Bounding Region (SBR)). *For a given group S of n time series data objects of length L, $S = \{s_1, s_2, ..., s_n\}$, the Skyline Bounding Region of S specifies a two-dimensional region surrounded by top and bottom skylines, and two vertical lines connecting the two skylines at the start and end times. The top (TSky) and bottom (BSky) skylines of S are defined as follows.*

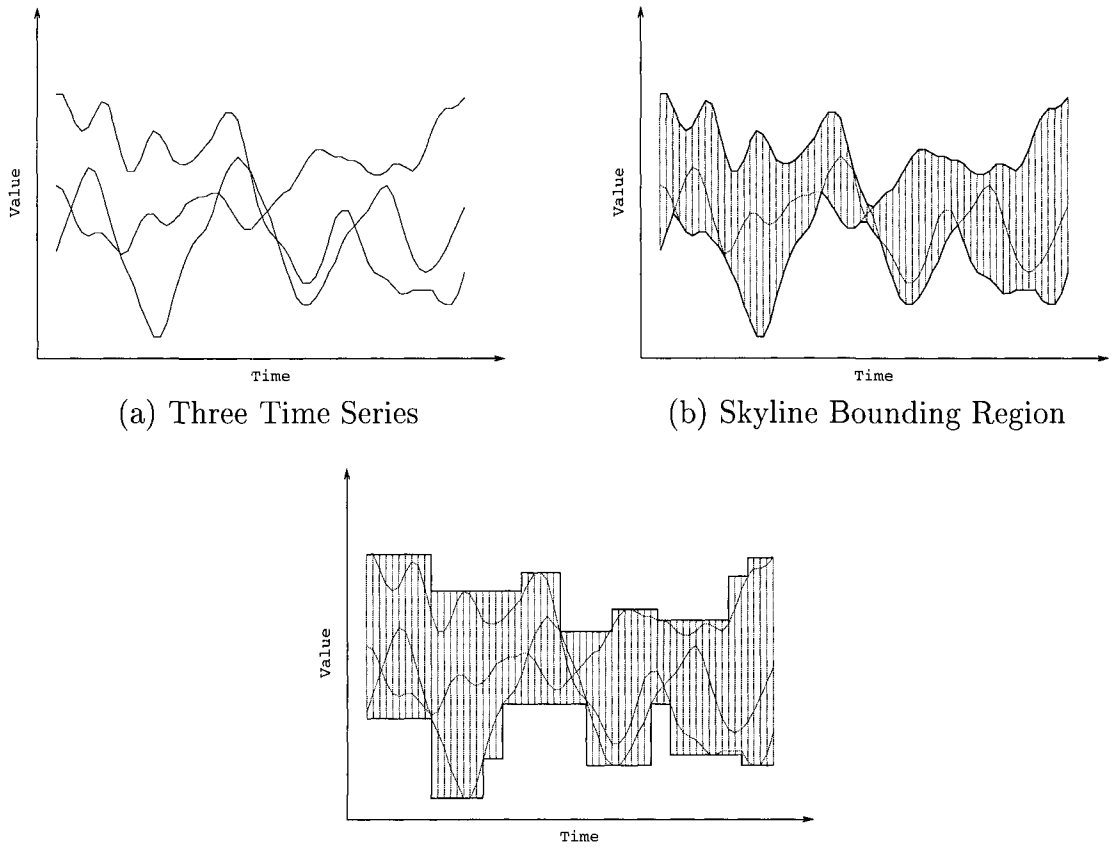
$$TSky = \{ts_1, ts_2, \ldots, ts_L\}, \qquad BSky = \{bs_1, bs_2, \ldots, bs_L\},$$

where, for $1 \leq i \leq L$,

$$ts_i = max\{s_1[i], \ldots, s_n[i]\} \quad and \quad bs_i = min\{s_1[i], \ldots, s_n[i]\},$$

and $s_j[i]$ is the i_{th} value of the j_{th} time series data in S.

Note that, unlike MBRs generated from the APCA representation, Skyline Bounding Regions (SBRs) are composed of only one region. Therefore SBRs are free of internal overlap. As an example, Figure 5.2(b) shows the skyline bounding region for the three time series data in Figure 5.2(a). The straightforward adoption of the SBR definition is impractical, because its representation can be too long to be efficiently handled by an index (*i.e.*, 256 or longer for our experiments). Therefore, only approximate representations of $TSky$ and $BSky$ should be stored for each SBR in an index structure. Care should be taken to ensure that the region defined by the approximate SBR encloses that of the original SBR. This will ensure the *group lower-bound property* (second condition of Lemma 1) is met by the skyline distance function, which will be introduced next. Two good candidates for approximating the skylines are to use either equal-length or variable-length constant-valued segments, in a similar way to Segmented Means [131] and Adaptive Piecewise Constant Approximation [74]. Among these candidates, it has been shown that the APCA representation has the smallest approximation error [74]. Thus we have chosen to use APCA to approximate SBRs in the Skyline index. An example of variable-length constant-valued segments



(a) Three Time Series (b) Skyline Bounding Region

(c) Approximate SBR

FIGURE 5.2. Skyline Bounding Region of Time Series Data

approximating a skyline bounding region is shown in Figure 5.2(c). For the rest of this chapter, we assume the APCA representation is used to approximate Skyline Bounding Regions.

5.2.2 Quality of the Bounding Regions

In this section, we compare the Skyline index with the APCA index for the quality of indexing. We base our comparisons on the differences between the bounding regions defined by the indexes (*index bounding regions*) and the bounding regions defined by raw data (*data bounding regions*). If we think of a time series data as an L-dimensional

Segments	$Q_{Apca}(I_R)$	$Q_{skyline}(I_R)$
8	1.673	1.326
16	1.700	1.345
32	1.700	1.337

TABLE 5.3. Comparing the Quality of the APCA and Skyline Bounding Regions

vector, the *data bounding region* for a group, G, of L-dimensional vectors is the minimal L-dimensional bounding hyper-rectangle for G. We estimate the quality of the index by the ratio of the area covered by the *index bounding region* (in the *time-value* space) to the area covered by the *data bounding region* as follows.

$$Q(I_R) = \frac{Area(IndexBoundingRegion)}{Area(DataBoundingRegion)}, \tag{5.1}$$

where I_R is an index node for G, $IndexBoundingRegion$ is the bounding region, R, defined by the index for node I_R, and $DataBoundingRegion$ is the bounding region defined by G. For the Skyline index, this ratio is defined as

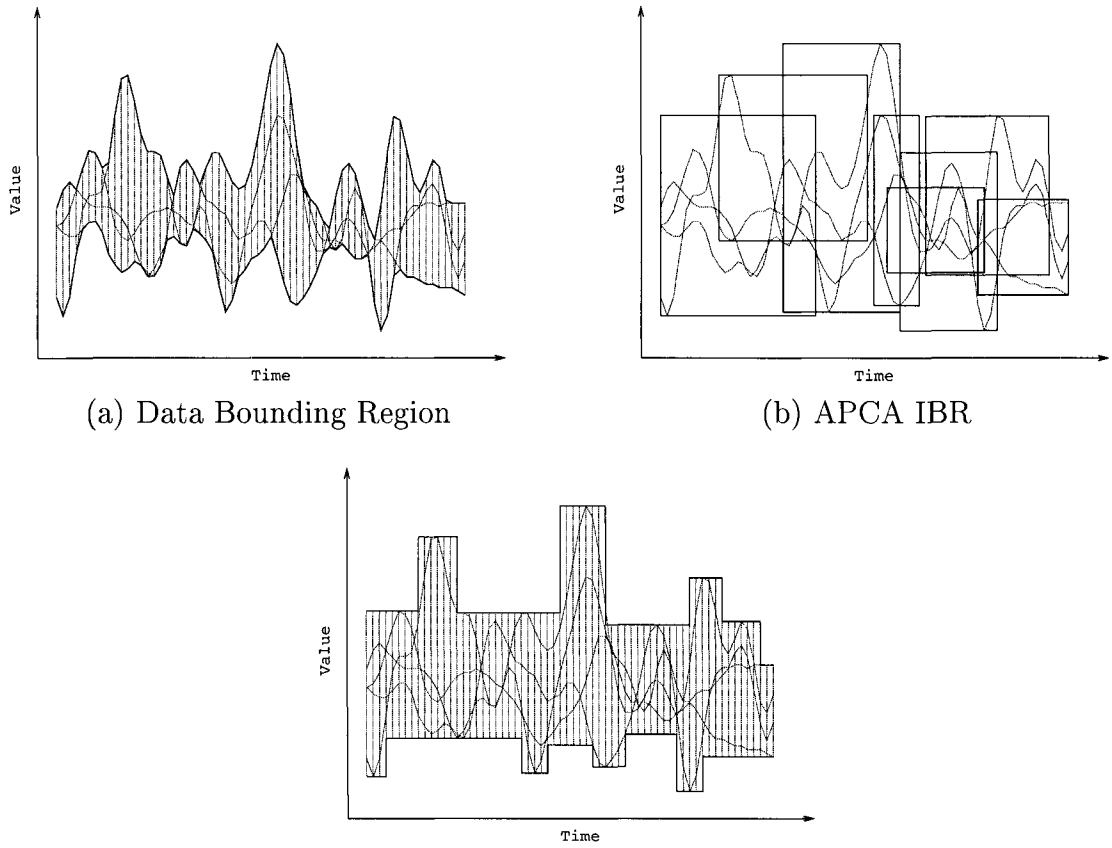
$$Q_{skyline}(I_R) = \frac{Area(ApproximateSBR)}{Area(DataBoundingRegion)}.$$

For the APCA index, we use

$$Q_{Apca}(I_R) = \frac{Area(\cup_{i=1}^{M}(R_i))}{Area(DataBoundingRegion)}, \quad \{R_1, R_2, \ldots, R_M\} = R,$$

where R is the set of M 2-dimensional rectangles defined by the MBR for node I_R in the APCA index.

In our preliminary experiments with the EEG data set, we computed these ratios for both skyline and APCA bounding regions. For each method, we built three indexes using 8, 16, and 32 segments (*i.e.*, 16, 32, and 64 dimensions), respectively. In this experiment, both the APCA and the Skyline index are based on the R-tree. We traversed each index and computed the quality of the bounding region for each index node. Table 5.3 shows the average results from our comparisons. We observed that the area of a bounding region as defined by the APCA index is 70% larger than



(a) Data Bounding Region

(b) APCA IBR

(c) Skyline IBR

FIGURE 5.3. Data and Index Bounding Regions (IBR)

the bounding region defined by the raw data. The Skyline index, on the other hand, defined smaller bounding regions, only 33% larger than the regions defined by the raw data. Remember that the bounding region defined by raw data is minimum. Therefore, we used the area of the data bounding region as a yardstick in this experiment. These results suggest that the use of skyline bounding regions will result in a more efficient index. In Figure 5.3, we show an example of data bounding region (a), APCA index bounding region (b), and Skyline index bounding region (c). From this figure, it becomes clear that the skyline technique provides a more accurate representation of a data bounding region than the APCA.

5.2.3 Skyline Distance Function

In order to use the SBR representation in a multi-dimensional index, we must have a distance function $D_{Region}()$ that lower-bounds the distance between a query object and a group of time series data. Keogh [69] defined a distance function $LB_{Keogh}()$ to lower-bound the Dynamic Time Warping distance between a data time series and an envelope containing all possible temporal shiftings of the query time series given a constraint on the warping path. In the Skyline index, we have a set of data time series contained within an SBR (a kind of envelope). Therefore, we can use $LB_{Keogh}()$ to lower-bound the distance from a query object to an SBR. Formally, given a query $\overline{q}$ and an SBR R (*e.g.*, an index node) that contains a group of time series data, the lower-bounding distance function $LB_{Keogh}()$ is defined as follows.

$$LB_{Keogh}(\overline{q}, R) = \sqrt{\sum_{i=1}^{L} (dist(\overline{q}[i], TSky[i], BSky[i]))^2} \tag{5.2}$$

where $\overline{q}[i]$ is the i_{th} value of the query $\overline{q}$, L is the length of each time series data, and

$$dist(q_i, t_i, b_i) = \begin{cases} q_i - t_i & \text{if } q_i > t_i \\ b_i - q_i & \text{if } q_i < b_i \\ 0 & \text{otherwise.} \end{cases}$$

The following lemma shows that $LB_{Keogh}()$ satisfies the *group lower-bound property.*

Lemma 2. *Given a query $\overline{q}$ and any time series data $\overline{x}$ contained in an SBR R,*

$$LB_{Keogh}(q, R) \leq \|\overline{q} - \overline{x}\|_2, \ \forall \overline{x} \in R \tag{5.3}$$

where $\|\overline{q} - \overline{x}\|_2$ represents the Euclidean distance between $\overline{q}$ and $\overline{x}$.

Proof. For any $\overline{x} \in R$, a value $\overline{x}[i]$ at time $i\,(1 \leq i \leq L)$ is bounded by $TSky[i]$ and $BSky[i]$. Thus,

$$(dist(\overline{q}[i], TSky[i], BSky[i]))^2 \leq (\overline{q}[i] - \overline{x}[i])^2.$$

From this, it is obvious that $LB_{Keogh}(\overline{q}, R)$ satisfies the group lower-bound property. □

Note that Equation (5.2) and Lemma 2 have been described based on full SBR representations (*e.g.*, Figure 5.2(b)) for simpler presentation. The $LB_{Keogh}()$ distance function can also be defined for approximate SBR representations (*e.g.*, Figure 5.2(c)), and Lemma 2 will still be satisfied.

5.2.4 Indexing Time Series

Following the GEMINI [42] paradigm, the approximate representations of time series data can be indexed by a spatial access method. We implemented the Skyline index based on the R-tree structure [56]. In an R-tree based *Skyline index*, each entry in an internal node consists of the approximate representation of an SBR and a pointer to a child node. An entry in a leaf node, on the other hand, consists of the approximate representation of and a pointer to a time series data object. Note that the *Skyline index* is not restricted to a particular approximation to represent time series data objects. Since it is possible to calculate the SBR of a leaf node based on raw time series data, we can use any approximation method as long as the corresponding $d_{feature}()$ function satisfies the *contractive property* (condition 1 in Lemma 1). Alternatives for approximating time series data are to use variable-length constant-valued segments, equal-length constant-valued segments, or even DFT or DWT transformations. We have chosen to use variable-length constant-valued segments (*i.e.*, APCA) to represent time series data because of the high fidelity of its approximation [74].

The insertion algorithm used by the *Skyline index*, which is described below, is similar to the R-tree insertion algorithm proposed by Guttman [56].

step 1. *Find a position for a new record.* Starting from the root node, traverse the tree to find the best leaf node for inserting the new entry.

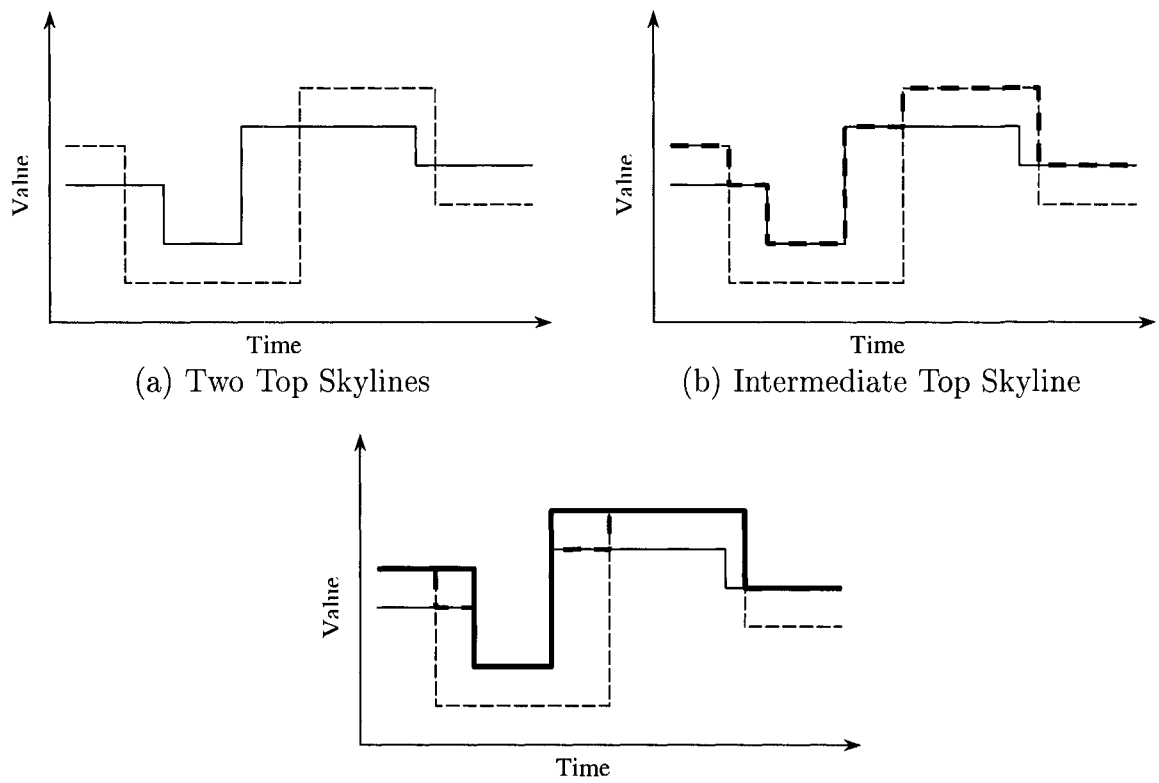
step 2. *Add the record to a leaf node.* Split the node if it is full.

step 3. *Propagate changes upward.* Ascend the tree from the leaf node. Adjust SBRs and propagate node splits as necessary.

step 4. *Grow the tree taller.* If propagation caused the root to split, create a new root whose children are the two resulting nodes.

During **step 1**, at each level of the tree, *Skyline index* selects the node whose approximate SBR requires the least area enlargement to accommodate the new entry. An approximate SBR is enlarged if, for a given segment, a value in the new entry is outside the limits defined by the segment. In such case, the value of the segment is updated to completely contain the new entry. At **step 2**, if a split is needed, entries are re-distributed by Guttman's quadratic algorithm [56] to minimize the area of the resulting SBRs.

In a hierarchical index structure, the SBR of a parent node is determined from the SBRs of its child nodes, such that the parent SBR minimally encloses all the child SBRs. In **step 3**, some SBRs may need modifications to accommodate a new entry. Here, we describe how to expand the SBR of an internal node. Later, we describe how to obtain the SBR of a leaf node during insertion. An SBR consists of a top and bottom skyline. Since expanding the bottom skyline is symmetrical, we only focus on describing how the top skyline is expanded. We start by merging the top skylines of the approximate SBR of an internal node and a new entry. Figure 5.4(a) shows an example of two skylines to be merged. Each skyline has four segments. We scan



(a) Two Top Skylines

(b) Intermediate Top Skyline

(c) Resulting Top Skyline

FIGURE 5.4. Merging Two Skylines

the two skylines together from left to right along the time dimension generating a new intermediate top skyline (Figure 5.4(b)). If the resulting number of segments is larger than required, we iteratively coalesce pairs of consecutive segments until we get the required number of segments. Two consecutive segments $<v_i, r_i>$ and $<v_{i+1}, r_{i+1}>$ are coalesced into $<v'_i, r_{i+1}>$, where $v'_i = max\{v_i, v_{i+1}\}$. Note that, in a top skyline segment $<v, r>$, v is the maximum value and r is the right end of the segment. The pair of consecutive segments that results in the minimum increase of the approximation error is chosen in each coalescing operation. Figure 5.4(c), shows the final result from merging the skylines in Figure 5.4(a). After coalescing, only 4 segments are left in the resulting skyline.

One concern about the iteratively coalescing of pairs is that the quality of the ap-

proximate SBR for an index node (U) may degrade after consecutive insertions. Consequently, the resulting approximate SBR, which we refer to as the *merge-approximate SBR*, may not tightly approximate the SBR defined by the time series data indexed by U. We conducted one more preliminary experiment using the EEG data set to investigate this issue. We built three Skyline indexes using 8, 16, and 32 segments, respectively. Each index was incrementally built by inserting the 99,000 entries described previously. For each index node U, we fetched all time series data indexed by the sub-tree rooted at U. The collective shape of this set of time series data defines the SBR for U. We approximated this SBR (*e.g.*, using APCA) resulting in the *direct-approximate SBR* for U. Note that this direct-approximate SBR was generated from the SBR defined by raw time series data. Therefore, it is free of any quality degradations due to the merging mechanism used to process insertions. We compared the direct-approximate SBR of U to its *merge-approximate* SBR (*i.e.*, the approximate SBR for U stored in the index). To measure the degradation of the index SBR due to the merging process, for each index node, we computed the quality of both its *direct-approximate* and *merge-approximate* SBR using Equation (5.1) given in Section 5.2.2. We present the average results of our measurements in Table 5.4, labeled as Q_{direct} and Q_{merge}, respectively. These results show that the quality degradation of the approximate SBRs after consecutive insertions is insignificant (*i.e.*, less than 5%) with respect to the direct-approximate SBR. This gave us an indication that the proposed merging mechanism is a viable technique to incrementally maintain approximate SBRs in the Skyline index. We should note that we used the same sub-optimal implementation of the APCA as in [74] to generate the direct-approximate SBRs. This had an adverse effect on the quality of the approximation which was more evident for approximations using a small number of segments. As it is shown in the first row of Table 5.4, when only 8 segments are used, the merging error introduced by the index was smaller than the error introduced by the sub-optimal approximation.

Segments	Q_{merge}	Q_{direct}	Q_{merge}/Q_{direct}
8	1.326	1.329	0.998
16	1.345	1.326	1.014
32	1.333	1.280	1.041

TABLE 5.4. Quality of Merge-Approximate SBR vs. Direct-Approximate SBR

5.2.5 Approximate SBRs in Leaf Nodes

We have mentioned before that the Skyline index is not restricted to a specific data approximation to represent time series. For example, we could approximate time series data using variable-length constant-valued segments, equal-length constant-valued segments, or even DFT or DWT transformations. Regardless of the data approximation of choice, we can trivially generate the SBR for a leaf node by retrieving all the time series data indexed by the node. Once we have obtained the SBR, we can approximate it and then merge it into the index as described before (*i.e.*, step 3 of insertion algorithm).

One problem with this approach is a poor performance due to the excessive IO required by each insertion (inserting a new entry in leaf node U requires fetching all time series data contained in U). Instead, we adopt an alternative solution in which a new time series data entry is considered to be an SBR. A data time series of length L is an SBR with L segments in which its top and bottom skylines are equal. Because of the high cost of merging L segments, we opt for merging an approximation of the time series using a smaller number of segments. The approximation of this SBR can be easily obtained. For example, we could use the APCA approximation, in which case the *min* and *max* values of each segment define the top and bottom skyline of the new entry. After this, expanding an leaf node SBR is done in the same way the SBR for an internal node is expanded (described before).

5.2.6 Skyline k-Nearest Neighbor Search

The Skyline index can be used with Algorithm 4 to provide k-Nearest Neighbor (k-NN) Search. We need to substitute the generic functions $d_{feature}()$ and $D_{Region}()$. If we choose APCA as the approximation for time series data, the function $d_{Apca}()$ replaces $d_{feature}()$. The $d_{Apca}()$ distance function lower-bounds the Euclidean distance between a query and a data time series. Therefore, condition 1 of the Lemma 1 is satisfied. Also, $D_{Region}()$ will be replaced by $LB_{Keogh}()$, which guarantees the *group lower-bound property* of Lemma 1. It should be clear that Algorithm 4 using Skyline index will produce correct answers.

5.2.7 Lower-bound Distances and Search Performance

In this section, we state the theoretical basis for the argument that the use of skyline bounding regions improves the k-NN search performance. In particular, we investigate the combined effect of the lower-bound distance function and the index bounding regions on the number of candidates retrieved during k-NN search. We also study how these factors are related to the concept of r-optimality defined by Seidl and Kriegel [110].

An r-optimal multi-step k-NN algorithm must retrieve every candidate object c, whose feature distance from a given query $\overline{q}$ is less than or equal to the Euclidean distance to the k_{th} nearest neighbor $\overline{x}_k$ from $\overline{q}$. That is, $d_{feature}(\overline{q}, c) \leq \|\overline{q} - \overline{x_k}\|_2$, where $d_{feature}()$ is a lower-bounding distance function adopted by the multi-step k-NN algorithm. Let us call the set of all these objects retrieved by the r-optimal algorithm the *minimal candidate set.* This set is *minimal* because it *only* includes objects satisfying $d_{feature}(\overline{q}, c) \leq \|\overline{q} - \overline{x}_k\|_2$. It is generally assumed that the distance to a region R lower-bounds $d_{feature}(\overline{q}, x')$, $\forall \overline{x} \in R$. We formally define this property as the *containment property.* The containment and the contractive properties are *necessary conditions* for r-optimality.

Definition 6 (Containment Property). *For a given query $\overline{q}$ and a bounding region R (*e.g.*, an index node) containing a group of time series data,*

$$D_{Region}(\overline{q}, R) \leq d_{feature}(\overline{q}, x'), \ \forall \overline{x} \in R \tag{5.4}$$

where $D_{Region}(\overline{q}, R)$ represents a lower-bounding distance from $\overline{q}$ to the bounding region R, x' is the feature vector of $\overline{x}$, and $d_{feature}(\overline{q}, x')$ is a lower-bound of the distance from $\overline{q}$ to $\overline{x}$.

While the containment and the contractive(Lemma 1) properties are necessary conditions for a k-NN search algorithm to be r-optimal, the containment property is not a necessary condition for the *correctness* of the k-NN search algorithm. For instance, it has been shown that the APCA representation produces correct k-NN search results despite the fact that it does not satisfy the containment property [74]. The APCA representation does not satisfy the containment property because the $D_{Apca}()$ distance function is computed based on a definition of MBR, which is independent of the lower-bounding distance function $d_{Apca}()$. In particular, for a bounding region R, while $D_{Apca}(\overline{q}, R) \leq \|\overline{q} - \overline{x}\|_2$, and $d_{Apca}(\overline{q}, x') \leq \|\overline{q} - \overline{x}\|_2 \ \forall \overline{x} \in R$, there may exist an $\overline{x} \in R$ such that $d_{Apca}(\overline{q}, x') < D_{Apca}(\overline{q}, R)$. This is illustrated in Figure 5.5, which shows an example of an APCA MBR with one segment. In the figure, the mean values of time series $\overline{q}$ and $\overline{x}$ are the same(*i.e.*, $d_{Apca}(\overline{q}, x') = 0$). Since $\overline{q}$ is not enclosed by the MBR R, then $D_{Apca}(\overline{q}, R) > 0$, therefore, $D_{Apca}(\overline{q}, R) > d_{Apca}(\overline{q}, x')$. Clearly, this example illustrates that the *containment property* is not satisfied by the APCA representation.

For a similar reason, the containment property does not hold for $d_{Apca}()$ and $LB_{Keogh}()$ under the proposed Skyline index approach, and hence Algorithm 4 is not r-optimal either. However, it is still true that the search results will be correct because Lemma 1 holds for $d_{Apca}()$ and $LB_{Keogh}()$. We now show that the number of objects retrieved by this algorithm could actually be less than the number of elements

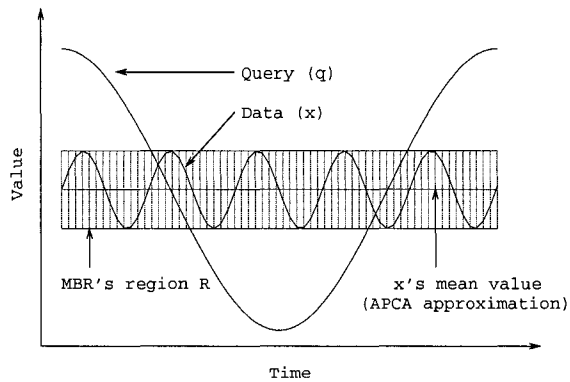


FIGURE 5.5. APCA does not Satisfy the Containment Property

in the *minimal candidate set* defined by $d_{Apca}()$ (the candidate set is minimal with respect to a lower-bounding distance function, changes to this function affect the size of the candidate set). Let us start by defining a *virtual* lower-bounding distance to be used by the Skyline index. Given a query $\overline{q}$, a data time series $\overline{x}$, and its APCA approximation x', we define a new lower-bounding distance function $d^v_{Apca}(\overline{q}, x')$ as follows.

$$d^v_{Apca}(\overline{q}, x') = \max\{d_{Apca}(\overline{q}, x'), LB_{Keogh}(\overline{q}, R)\}, \tag{5.5}$$

where R is the immediate bounding region (*i.e.*, direct parent) of $\overline{x}$. Because $d^v_{Apca}(\overline{q}, x')$ and $LB_{Keogh}(\overline{q}, R)$ now satisfy the containment property, Algorithm 4 becomes r-optimal with $d_{Apca}(\overline{q}, x')$ replaced by $d^v_{Apca}(\overline{q}, x')$ in Line 13. In general, Equation (5.5) can be defined as

$$d^v_{feature}(\overline{q}, x') = \max\{d_{feature}(\overline{q}, x'), D_{Region}(\overline{q}, R)\}. \tag{5.6}$$

Lemma 3 (Interchangeability). *In a multi-step k-NN search algorithm* (e.g., *Algorithm 4), two lower-bounding distance functions* $d_{feature}(\overline{q}, x')$ *and* $d^v_{feature}(\overline{q}, x')$ *as defined in Equation (5.6) can be used interchangeably without affecting the correctness of results and the number of candidates to be retrieved.*

Proof. First, we know that with either distance function, the k-NN algorithm described in Algorithm 4 will terminate when $\|\overline{q} - \overline{x}_k\|_2$ is smaller than the lower bound distance at the front of the priority queue, where $\overline{x}_k$ is the k_{th} nearest neighbor. Now, to prove that Lemma 3 is correct we only need to show that the following two assertions hold.

1. Any candidate accessed in the algorithm using $d_{feature}()$ is also accessed in the algorithm using $d^v_{feature}()$.

   Suppose $\overline{r}$ is any candidate accessed in the algorithm using $d_{feature}()$ and $\overline{r}$ is in MBR R. Since $\overline{r}$ has been accessed in the algorithm using $d_{feature}()$, we know that MBR R has been dequeued and expanded (that is why $\overline{r}$ has been inserted in the main priority queue.). From this, we can get $D_{Region}(\overline{q}, R) < \|\overline{q} - \overline{x}_k\|_2$ and $d_{feature}(\overline{q}, \overline{r}) < \|\overline{q} - \overline{x}_k\|_2$. According to the definition of $d^v_{feature}()$, we have $d^v_{feature}(\overline{q}, \overline{r}) < \|\overline{q} - \overline{x}_k\|_2$. Thus, $\overline{r}$ should have been accessed in the algorithm using $d^v_{feature}()$.

2. Any candidate accessed in the algorithm using $d^v_{feature}()$ is also accessed in the algorithm using $d_{feature}()$.

   Suppose candidate $\overline{r}$ is accessed in the algorithm using $d^v_{feature}()$. Since $\overline{r}$ has been dequeued from the main priority queue, we have $d^v_{feature}(\overline{q}, \overline{r}) < \|\overline{q} - \overline{x}_k\|_2$. From the definition of $d^v_{feature}()$, we can get $D_{Region}(\overline{q}, R) < \|\overline{q} - \overline{x}_k\|_2$ and $d_{feature}(\overline{q}, \overline{r}) < \|\overline{q} - \overline{x}_k\|_2$. If $D_{Region}(\overline{q}, R) < \|\overline{q} - \overline{x}_k\|_2$, then MBR R should have been dequeued and expanded in the algorithm using $d_{feature}()$, and $\overline{r}$ should have been inserted in the main priority queue. Since $d_{feature}(\overline{q}, \overline{r}) < \|\overline{q} - \overline{x}_k\|_2$, $\overline{r}$ should also have been accessed in the algorithm using $d_{feature}()$ before it ends.

From the above two assertions, we know that both algorithms access the same set of candidates. If they access the same set of candidates, they produce the same results. Therefore, the two algorithms are equivalent. □

Since the virtual function $d^v_{feature}(\overline{q}, x')$ is always larger or equal to $d_{feature}(\overline{q}, x')$, the minimal candidate set defined by the virtual function can be smaller than that defined by $d_{feature}(\overline{q}, x')$. The implication of Lemma 3 is exceedingly important. It provides the theoretical basis of the argument that the use of *skyline bounding regions (SBR)* can reduce the number of candidates to be retrieved for k-NN search. By Lemma 3, as long as the group lower-bound property is satisfied, different techniques can be used to group data into hierarchical index structures. Instead of using the default grouping method provided by the R-tree, one can choose the best grouping technique with tighter bounds, while keeping the correctness of the k-NN search algorithm. Thus, if one can represent time series data with tighter bounding regions than those being used, either by adopting a different approximation or an entirely different index organization, the overall performance of k-NN search can be improved by reducing the number of index accesses and the number of candidates to be retrieved. This is precisely the way we can improve k-NN query processing by adopting the skyline bounding regions.

5.3 Performance Evaluation

In this section, we empirically demonstrate the improved performance the proposed *Skyline Index* approach over state-of-the-art techniques such as APCA. In addition, we observe the impact of different distance functions by different dimensionality reduction and indexing techniques on the overall similarity search performance. We performed our comparisons in the context of k-NN similarity search. We studied combinations of the Haar wavelet transformation (DWT) with the M-tree [31] and R-tree. We performed the same study for the APCA data approximation and included the Hybrid-tree [20], in addition to the M-tree and R-tree. During our empirical evaluation, we used three different metrics to assess the performance of each of the techniques included in this study. In particular we used *index search overhead*,

number of data objects fetched, and *total elapsed time.*

5.3.1 Data Sets

We collected time series data from various sources of real-world applications and a synthetic data generator, and placed them into four separate data sets. Then, for each data set, we created three different cases by chopping each time series data into segments of length 1024, 512 or 256. Each resulting data set contains 100,000 time series objects of the same length. We randomly extracted 1% of the entries from each data set. This subset of 1,000 time series objects became our query set. We did this to avoid exact matches with queries in our experiments. In addition, this practice allowed us to use a query object that is in the same domain as the data set. Details of our testing data are provided next.

(1) *Mixed Bag:* This data was generated from four data sources: Space Shuttle data, Arrhythmia, Random Walk, and Exchange rate. These data sources are the same used in Keogh's work [74] and were obtained directly from the authors. A sliding window of step one was used to chop the time series provided by each data source into equi-length segments. The segmented time series data were normalized to have a mean of zero and standard deviation of one. The data set we used in our experiments is the union of all the time series obtained in this way.

(2) *Mixed S10:* This data set is identical to **Mixed Bag** except that a sliding window of step ten was used to generate segments.

(3) *ECG:* This data set is the electrocardiogram data from the MIT-BIH database distribution [87]. This data set contains 100,000 segmented time series data generated by a sliding window of step one.

(4) *FinTime:* This data set contains a set of stock data synthetically generated by a financial time series benchmark [63]. These synthetic data represent stock values at opening and closing times as well as the highest and lowest value of each day. We generated stock values for 100,000 companies for a period of 256, 512, and 1024 days.

5.3.2 Performance Metrics

In our experiments, we evaluated the efficiency of different techniques using three metrics. We measured *index search overhead* and the number of *data objects fetched* as the two main factors affecting overall performance of similarity search. In addition, we measured *elapsed time* as the performance metric directly perceived by the user.

(1) *Index search overhead:* This is the number of index page accessed during k-NN search. Ideally, during search only a small fraction of the index pages are retrieved. Unfortunately, due to the curse of dimensionality, the effectiveness of an index degrades as the dimensionality increases.

(2) *Data objects fetched:* It has been suggested that we should prevent performance bias due to disparities in the quality of implementation of the methods being compared [71]. Following this recommendation, we evaluated the effectiveness of different k-NN search methods by the number of *data objects fetched* from the database. The number of data objects fetched during a k-NN query represents a performance metric that is independent of the quality of implementation.

(3) *Elapsed time:* We used wall-clock time to measure the elapsed time during the evaluation of k-NN queries. This time includes both CPU and IO time. In addition, for each query we recorded the time spent on CPU operations only. This allows us to estimate the time spent in IO operations. Finally, to avoid

any caching effects on the index and data files, we flushed the main memory between consecutive queries by loading irrelevant data into the system.

Testing and benchmarking were performed on Intel Pentium workstations running Linux kernel version 2.4.7. Each workstation has 600 MHz clock rate, 128 MBytes of main memory and 9 GBytes of disk storage with SCSI interface.

For each presented measurement, we ran 100 queries, and averaged the results from those 100 runs. Throughout the experiments, we measured the performance of k-nearest neighbor queries processed by different index techniques.

5.3.3 Evaluated Techniques

To show the performance advantages of our proposed Skyline index, we compare it to some of the widely accepted techniques for k-NN search for time series data. We used two different dimensionality reduction techniques in our experiments: the Haar wavelet transformation (DWT) and APCA. We built indexes based on the H-tree, R-tree, and M-tree.

(1) *Hybrid-tree:* We used the Hybrid-tree [20] to replicate the experiments described by Keogh *et al.* [74]. In this work, Hybrid-tree indexes were built in memory to simulate index IO operations by counting the number of accessed nodes. Since we were interested in measuring the overall performance of similarity queries, including the time spent during IO operations, we did not use the Hybrid-tree in combination with other methods. The source code of the hybrid tree was obtained from `http://www-db.isc.uci.edu`.

(2) *R-tree:* The Skyline index was implemented using the R-tree interface provided by the GiST [60] library. We extended the R-tree GiST interface and provided two new objects; *Skyline Bounding Region* and *APCA point.* Skyline bounding regions are used for handling approximated SBRs in internal nodes, whereas

APCA points are used for handling data approximation in leaf nodes. The actual implementation of the Skyline index using a popular index library such as GiST demonstrates the practical feasibility of the idea of using skyline bounding regions for indexing time series data. We also used GiST for implementing the R-tree index in combination with the APCA and DWT approximations. This gave us a common implementation framework for fair performance comparisons.

(3) *M-tree:* We include the M-tree in this study to provide a framework for comparing different index organizations and to illustrate the effect that different bounding regions have in the k-NN search performance. In the M-tree, a bounding region S is defined by a hyper-sphere centered at a routing object o_c with a radius γ. The radius γ is the maximum Euclidean distance to any object in S from the routing object o_c (*i.e.*, $\|\overline{s} - o_c\|_2 \leq \gamma,\ \forall \overline{s} \in S$). Thus, for any time series data $\overline{s} \in S$ and a query $\overline{q}$,

$$\begin{aligned} \|\overline{q} - \overline{s}\|_2 &\geq \|\overline{q} - o_c\|_2 - \|\overline{s} - o_c\|_2 \\ &\geq \|\overline{q} - o_c\|_2 - \gamma \\ &\geq d_{Apca}(\overline{q}, o_c) - \gamma \end{aligned}$$

because $d_{Apca}(\overline{q}, o_c)$ lower-bounds $\|\overline{q} - o_c\|_2$. Therefore, we can define a lower-bounding distance for S as follows.

$$D_{Sphere}(\overline{q}, S) = max\{d_{Apca}(\overline{q}, o_c) - \gamma, 0\} \tag{5.7}$$

Now, let us describe how different indexing mechanisms were combined with the dimensionality reduction techniques in our experiments. In the following, we summarize all the techniques we evaluated. In the description, we use $\overline{q}$ and $\overline{x}$ to represent query and data time series, respectively, and R to represent a bounding region.

(1) *Skyline:* We implemented the Skyline index based on the R-tree (implemented by the GiST library [60]). The $LB_{Keogh}(\overline{q}, R)$ was used to lower-bound $\{\|\overline{q} - \overline{x}\|_2$

$| \forall \overline{x} \in R\}$. The APCA representation was used to approximate individual time series data and $d_{Apca}(\overline{q}, x')$ was used to lower-bound $\|\overline{q} - \overline{x}\|_2$. We approximated time series data in APCA representations followed the steps suggested in Keogh's work [74].

(2) *Hybrid-tree APCA:* This is the same implementation as described in [74]. The Hybrid-tree [20] was used to index time series data in the APCA representation. The $d_{Apca}(\overline{q}, x')$ and $D_{Apca}(\overline{q}, R)$ were used to lower-bound $\|\overline{q} - \overline{x}\|_2$ and $\{\|\overline{q} - \overline{x}\|_2 \mid \forall \overline{x} \in R\}$, respectively.

(3) *R-tree APCA:* Same as **Hybrid-tree APCA** except that the GiST R-tree was used instead of the Hybrid-tree.

(4) *M-tree APCA:* The distance function $D_{Sphere}()$, defined by equation 5.7, was used in M-tree. Note that $d_{Apca}()$ does not satisfy the triangular inequality [74]. Therefore, we cannot build an M-tree based on such a definition of distance. To overcome this problem, we measured the radius of a hyper-sphere using the Euclidean distance defined by raw, instead of approximated, time series data (but only data approximations were inserted into the index). The $d_{Apca}(\overline{q}, x')$ distance function was used to lower-bound $\|\overline{q} - \overline{x}\|_2$.

(5) *R-tree DWT:* We implemented the Haar wavelet transformation [21] using the GiST R-tree index.

(6) *M-tree DWT:* The Haar wavelet transformation and its feature distance function were used with the M-tree index. To make it comparable to **M-tree APCA**, we measured the radius of a hyper-sphere by Euclidean distance, and used a distance function similar to $D_{Sphere}(\overline{q}, R)$ to lower-bound $\{\|\overline{q} - \overline{x}\|_2 \mid \forall \overline{x} \in R\}$.

All indexes were built using pages of 8 KBytes. All values in time series data were stored in 8-byte long `double` format.

5.3.4 Experimental Results

In our experiments, we extensively evaluated the performance of the different k-NN search techniques described in Section 5.3.3. We measured the performance of these techniques using different data sets, different values of k, and different lengths of the time series.

Index Search Overhead and Data Objects Fetched: Tables 5.5 and 5.6 show the IO performance results obtained from our experiments. Each table shows the number of *data objects fetched* by each of the techniques evaluated in this study for k-NN queries (Tables 5.5(a) and 5.6(a)). We remind the reader that in a k-NN query, data must be fetched to refine the candidate set and eliminate false alarms. Therefore, the best performance will be shown by the approach that requires fetching less data objects. This measure is free of implementation bias allowing direct comparisons across all the evaluated techniques. These tables also show the *index search overhead* of each technique as the number of index pages accessed during search (Tables 5.5(b) and 5.6(b)). In Table 5.5, we show results for 1-NN queries on three data sets with time series of length 1024. Table 5.6 shows results for the remaining data set for 10-NN queries and time series of length 256, 512, and 1024.

Several observations can be made based on the results shown in Tables 5.5 and 5.6. First, as expected, there was a clear trade off between data and index IO depending on the dimensionality of the feature vector used to approximate time series. The number of data objects fetched decreased as we increased the dimensionality of the data approximation. The reason is that, with a higher dimensionality, the fidelity of approximation improves thereby reducing the number of false alarms. On the other hand, increasing the dimensionality of the approximation had a negative effect on the index performance. As the dimensionality increased, the number of index pages accessed during search also increased.

Second, we also observed that the choice of index organizations had non-trivial

Data set	Dim	Number of Data Objects Fetched					
		H-tree APCA	**R-tree APCA**	**R-tree DWT**	**M-tree APCA**	**M-tree DWT**	**Skyline**
ECG	16	158	156	3,678	157	3,678	155
	32	10	8	565	9	565	8
	64	3	2	48	2	48	2
Mixed S10	16	3,270	2,480	1,103	3,133	1,103	1,708
	32	1,819	1,462	1,010	1,679	1,010	863
	64	1,299	998	986	1,218	986	439
FinTime	16	1,915	1,914	94	1,914	94	1,913
	32	78	77	17	77	17	77
	64	14	14	5	14	5	14

(a) Number of Data Objects Fetched

Data set	Dim	Number of Index Pages Accessed					
		H-tree APCA	**R-tree APCA**	**R-tree DWT**	**M-tree APCA**	**M-tree DWT**	**Skyline**
ECG	16	2,489	3,759	1,137	4,239	4,239	3,086
	32	4,938	6,379	1,350	7,277	7,278	3,150
	64	10,457	10,842	1,210	14,593	14,621	2,033
Mixed S10	16	2,444	2,415	1,223	1,527	1,396	804
	32	4,560	4,138	2,359	2,277	2,141	1,156
	64	9,887	7,983	4,669	3,957	3,789	1,665
FinTime	16	2,500	1,670	1,875	2,292	2,200	1,345
	32	4,947	3,409	3,874	3,845	3,771	2,259
	64	10,500	6,806	7,496	6,900	6,833	4,142

(b) Number of Index Pages Accessed

TABLE 5.5. IO Performance for 1-NN Queries on Time Series of Length 1024

impact on index overhead. Consider, for example, the Haar wavelet transformation, where the data IO was identical for R-tree and M-tree indexes (due to the r-optimality of the k-NN algorithm as discussed in Section 5.2.7). The impact of the index organization on the index overhead is more evident for the ECG and Mixed Bag data sets, where the R-tree outperformed the M-tree by a wide margin. For the other two data sets, the R-tree still outperformed the M-tree when 16-dimensional feature vectors were used.

Data length	Dim	Number of Data Objects Fetched					
		H-tree APCA	**R-tree APCA**	**R-tree DWT**	**M-tree APCA**	**M-tree DWT**	**Skyline**
256	16	26,292	17,769	15,890	25,907	15,890	12,072
	32	20,582	11,489	8,232	20,095	8,232	6,332
	64	15,859	8,487	5,090	14,980	5,090	2,970
512	16	24,866	15,185	16,928	24,528	16,928	11,077
	32	19,793	10,764	9,944	19,419	9,944	7,089
	64	14,503	7,925	5,772	14,054	5,772	3,460
1024	16	24,900	15,079	22,714	24,659	22,714	10,423
	32	22,954	11,686	20,022	22,899	20,022	7,805
	64	20,881	10,532	13,785	20,797	13,785	4,940

(a) Number of Data Objects Fetched

Data length	Dim	Number of Index Pages Accessed					
		H-tree APCA	**R-tree APCA**	**R-tree DWT**	**M-tree APCA**	**M-tree DWT**	**Skyline**
256	16	2,421	2,636	1,798	3,162	2,836	1,524
	32	4,918	4,617	3,478	4,886	4,145	2,292
	64	10,553	9,382	6,407	8,653	7,172	3,333
512	16	2,536	2,538	1,685	2,964	2,670	1,531
	32	5,010	4,535	2,866	4,637	4,001	2,254
	64	10,578	9,484	5,310	8,089	6,750	3,374
1024	16	2,521	2,460	1,175	2,768	2,676	1,505
	32	4,990	4,398	2,384	4,648	4,382	2,157
	64	10,708	9,479	4,819	8,353	7,393	3,479

(b) Number of Index Pages Accessed

TABLE 5.6. IO Performance for 10-NN Queries on the Mixed Bag Data Set

Third, If we compare the R-tree APCA and the Skyline indexes, we can observe the benefit of using skyline bounding regions on the index performance. Note that this is a valid comparison because both approaches are implemented on top of the R-tree interface provided by the GiST library. In both cases, the number of index pages accessed during k-NN queries increased as the dimensionality of the feature vectors increased. However, for the R-tree APCA, the number of index pages increased at a higher rate than for the Skyline index. This is consistent with our earlier observations

on the quality of the bounding regions defined by the APCA index. Regions defined by the APCA index suffer from internal overlap. The Skyline index, on the other hand, defines more efficient skyline bounding regions free of internal overlap. Additionally, we observed performance gains on the number of data objects fetched by the Skyline index with respect to the R-tree APCA. This result was consistent with the theoretical basis defined in Section 5.2.7. These performance gains were more evident for the Mixed Bag and Mixed S10 data sets.

Elapsed Time: In this section, we present the overall query performance as the elapsed time measured by wall-clock. We flushed the entire memory of the hardware system between consecutive runs of our experiments to avoid system caching effects. Furthermore, the portion of time used by the CPU was measured separately using the `clock()` function to profile the performance behaviors more accurately. We did not measure the elapsed time for *Hybrid-tree APCA*, because the Hybrid-tree implementation we obtained from the authors of the APCA work [74] pre-loaded an entire index into memory before starting query processing and made it impossible to measure time spent on actual IO operations.

Figures 5.6 and 5.7 show the performance results of five different methods measured in elapsed time for k-nearest neighbor queries. It was not surprising that the IO portion of the elapsed time followed closely the IO requirements shown in Tables 5.5 and 5.6. In general, the elapsed time spent on IO was the dominant factor of the overall execution time. In most of the cases, the Skyline index yielded the best overall performance. This was the result of the smallest IO requirements of the Skyline index both in number of data object fetched and in number of index pages accessed.

We also observed that *the CPU time cost was heavily affected by the choice of approximation methods.* With the same R-tree index, the APCA spent significantly higher amount of CPU time than the Haar transformation (DWT). This was due to the different computational requirements to compute lower-bound distance between

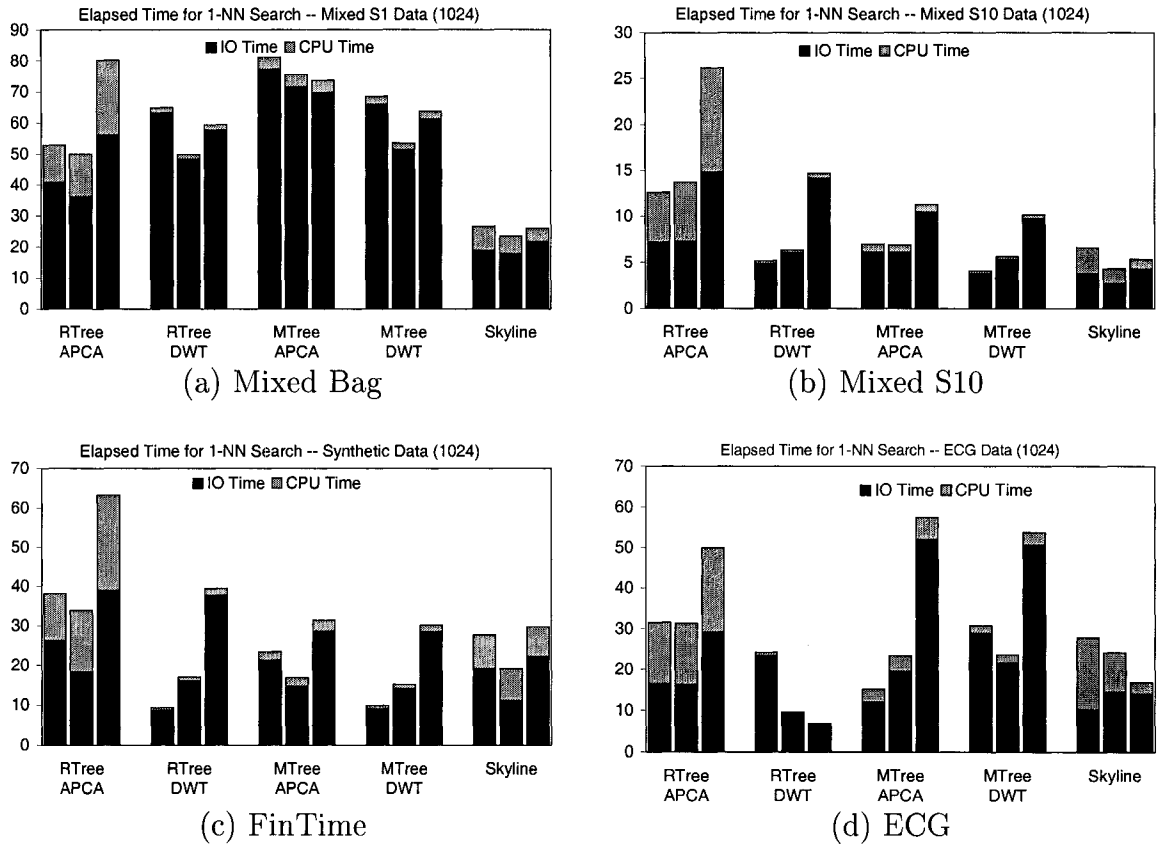
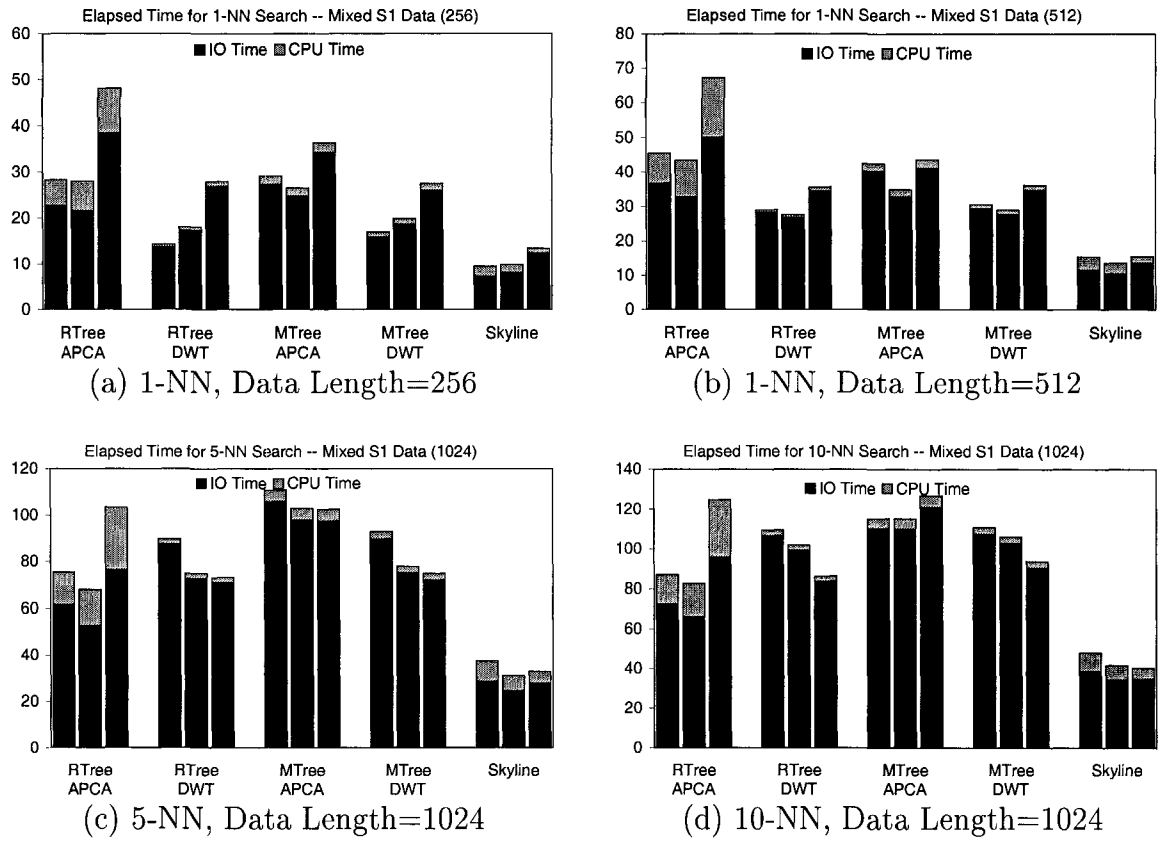


FIGURE 5.6. Elapsed Time, in Seconds, for 1-NN Queries on Time Series of Length 1024 (for each method, results for 16 (left), 32 (middle) and 64 (right) dimensional spaces). Overall Performance is Largely Influenced by IO

a query and a time series data. Under the Haar transformation, query and data time series are in the same representation and its distance can be computed without further transformation. Under the APCA representation, on the other hand, the APCA representation of a query q has to be regenerated from its raw data every time a lower-bound distance $d_{Apca}(\overline{q}, x')$ needs to be computed. In fact, the Skyline index also suffers from the high computational cost since $d_{Apca}()$ is used. Nonetheless, the Skyline index still outperformed the other methods in most of the cases due to its substantially reduced IO cost.

The same trend was observed from time series data of length 256 and 512 as



(a) 1-NN, Data Length=256 (b) 1-NN, Data Length=512

(c) 5-NN, Data Length=1024 (d) 10-NN, Data Length=1024

FIGURE 5.7. Elapsed Time, in Seconds, for k-NN Queries on the Mixed Bag Data Set (for each method, results for 16 (left), 32 (middle) and 64 (right) dimensional spaces). Overall Performance is Largely Influenced by IO

shown in Figure 5.7(a) and (b). Figure 5.7(c) and (d) show the elapsed time for 5- and 10-nearest neighbor queries, respectively. With an increased number of nearest neighbors, the dominance of the IO time in the overall performance grew even larger. This trend suggests that the Skyline index is anticipated to outperform the other methods with wider margins for larger k values.

5.4 Discussion

Now that we have provided detailed descriptions of our contributions in this paper, we proceed to make a brief comparison with previous work. To the best of our knowledge,

we are the first to use Skyline Bounding Regions (SBR) to organize and index time series data. An SBR is defined in the same *time-value* space where time series data are defined. Because of the large storage requirements of an SBR, the *Skyline index* only stores approximations of the SBR in the index nodes. Any approximation can be used to represent an SBR as long as the approximated SBR for an index node always bounds the time series data within the node. Previous work using approximations in the *time-value* space [70, 131], do not guarantee that the bounding region fully contains raw time series data. This is because their bounding regions are generated only to enclose the approximation of time series data, but not the data itself. To present concrete definitions, we used the L_2 norm as a similarity metric in this paper. However, since our index structure does not transform the *time-value* space, any L_p norm could be used.

5.5 Summary of Results

In this chapter, we introduce the *Skyline Index*, a simple and elegant paradigm for indexing time series data. The Skyline index uses the Skyline Bounding Region (SBR) to group time series data. To lower-bound the distance between a query time series and an SBR, the $LB_{Keogh}()$ function is used.

By analyzing the relationship between the feature distance of a time series data and the lower-bounding distance of its bounding region from a query, we show that the *containment property* is the necessary and sufficient condition of the r-optimality of multi-step k-NN search algorithms. Without the r-optimality, a multi-step k-NN search algorithm may have to retrieve a different number of candidates depending on the choice of feature representations and index organizations. We also show that the *group lower-bound property* is necessary to guarantee the correctness of k-NN search algorithm, but it cannot ensure retrieving the same set of candidates for different indexes, even if these indexes use the same feature representation.

Experimental results show that the Skyline Index can be coupled with the state of the art dimensionality reduction techniques and can improve the performance of similarity search by up to a factor of 3.

Chapter 6
A New Data Representation For Similarity Search on Time Series

In Chapter 4 we have observed that most of the existing techniques for the efficient evaluation of similarity search queries on time series follow the GEMINI paradigm [42]. That is, they extract small *signatures* to represent and approximate data objects by a data transformation. The data transformation should provide a mechanism to compute a lower-bound distance from a query to a data object only with its signature. This distance is used to estimate similarity between two time series. Typically, to speed up search operations, signatures are organized in a hierarchical index, such as the R-Tree [56]. While most of these transformations are ingenious ways for approximating data objects, they have not always succeeded in providing an efficient mechanism for processing similarity search queries. In fact, some of these methods are often outperformed by a simple linear scan on a data set.

The poor performance of these techniques can be attributed to two main factors: *index search overhead* and *data search overhead.* First, in a high-dimensional vector space, the overhead of accessing an index tends to be considerable, because a large portion of an index might need to be accessed. Second, performance is also affected by data search. During similarity search, it is required to fetch all data objects whose lower-bound distance to the query point is smaller than the search radius. The set of these qualifying objects is also known as the *candidate set* and its size depends on the quality of the lower-bound distance function provided by the approximation. A poor lower-bound distance function is more likely to cause a large number of objects to be retrieved. While the existing transformations approximate time series objects, they do not necessarily provide the tightest lower-bound distance.

To overcome these problems, we propose a new representation to approximate time series data that provides tight *upper-* and *lower-bound* distances. In addition, we avoid using hierarchical indexes for IO-efficient processing of similarity search. Our proposed method produces a compact representation of a time series data object. It takes advantage of the nature of time series data. That is, sequences of values do not follow a random distribution. Rather, these values originate from the same domain and follow patterns that can be identified. Our method adopts the idea of segmenting time series objects to identify subsequences of similar values. Differently from previous approaches, we adopt segmentation of time series data not only along the *time* dimension but also along the *value* dimension. We call this approximation *Self COntained Bit Encoding* (*SCoBE*) as it produces a quantized representation of a data object relative to the range of values found on its own segments. For IO efficient processing of similarity search, we organize the *SCoBE* approximations of a data set sequentially in a linear index.

The most significant characteristics of *SCoBE* can be summarized as follows.

- *SCoBE* provides a compact and highly accurate representation of time series data. It also provides both lower- and upper-bound distance functions.
- *SCoBE* is computed on individual basis. Given a time series object, there is no need to know the range of values in the entire data set to perform the encoding. Because of this, the existence of data outliers does not affect the quality of the approximation. This makes *SCoBE* an approximation that is more robust than previous quantization methods such as the VA-File [126] and the A-Tree [108].
- The storage overhead is minimized, since a linear index requires no more than the space needed to store the leaf nodes in a hierarchical index.
- While evaluating a similarity search query, the overhead for accessing *SCoBE*'s linear index is always constant. This linear index follows the same order as the

data. In addition, it is built with *SCoBE* approximations of the same size. This allows *SCoBE* to handle dynamic updates in constant time. To delete the i_{th} entry in the index, for example, we simply copy the last entry in the index to the i_{th} position and delete the last entry. For insertions, new entries are always added to the end of the linear index.

- The performance of similarity search using *SCoBE* is consistent across diverse data sets. It consistently outperforms sophisticated techniques such as the Adaptive Piecewise Constant Approximation [74]. The improvement in the observed performance is, in some cases, at least one order of magnitude.

- *SCoBE* can support any L_p norm as a measure of similarity.

The rest of this chapter is organized as follows. Section 6.1 presents the major limitations of previous approaches for the evaluation of similarity search queries on time series data. In Section 6.2, we introduce the *Self COntained Bit Encoding (SCoBE)* approach, whereas in Section 6.3, we discuss an improved algorithm for exact k-NN search using *SCoBE* organized in a linear index. In Sections 6.3.3 and 6.3.4, we show how *SCoBE* can be used for evaluating dynamic time warping and subsequence match queries. The results of experimental evaluation are presented in Section 6.4. Finally, Section 6.5 summarizes the contributions of this chapter.

6.1 Limitations of Previous Methods

The overall performance of similarity search is affected by the quality of the approximation, as well as by the index data structure used during the search. Keogh *et al.* [74] defined the *quality* of the approximation as the reconstruction error with respect to the original data. However, in the context of similarity search, the quality of the approximation should also be associated to the quality of the lower-bound distance function it provides.

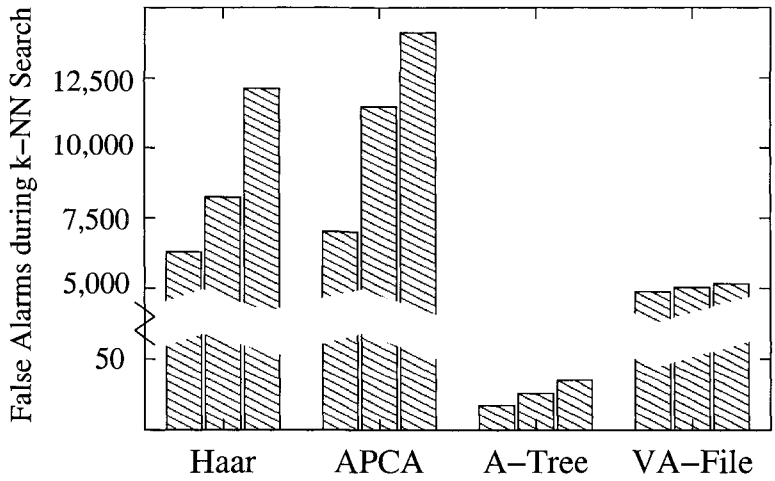


FIGURE 6.1. From Left to Right, Number of False Alarms for 5, 10, and 20 Nearest Neighbor Search. Fewer False Alarms Indicate a better Data Approximation

We conducted preliminary experiments to evaluate the quality of different approximations with respect to the k-NN search performance. For this experiment, we used the Mixed S10 data set described in Chapter 5 and we used the number of *false alarms* as our metric. The *false alarms* are irrelevant objects retrieved as candidates of the query result. To guarantee the exact answer to the query, every element in the candidate set must be fetched from the database and checked if it is a relevant part of the answer. Because each irrelevant object in the candidate set adds extra processing time for the query, we want to minimize the number of *false alarms*. Figure 6.1 shows the number of *false alarms* introduced by different approximation techniques during similarity search. The queries used were 5, 10, and 20-nearest neighbor searches. The number of time series objects in the database was 99,000. Each object in the database was a sequence of 256 values. The size of the data approximations was $\frac{1}{8}$ of the size of the data objects (*e.g.*, 16 segments for APCA, 32 coefficients for Haar). Comparisons using different sizes of approximations are presented in Section 6.4. In Figure 6.1, we observe that the A-Tree provided an approximation of higher quality than the other techniques. The quality of the A-Tree's approximation was reflected

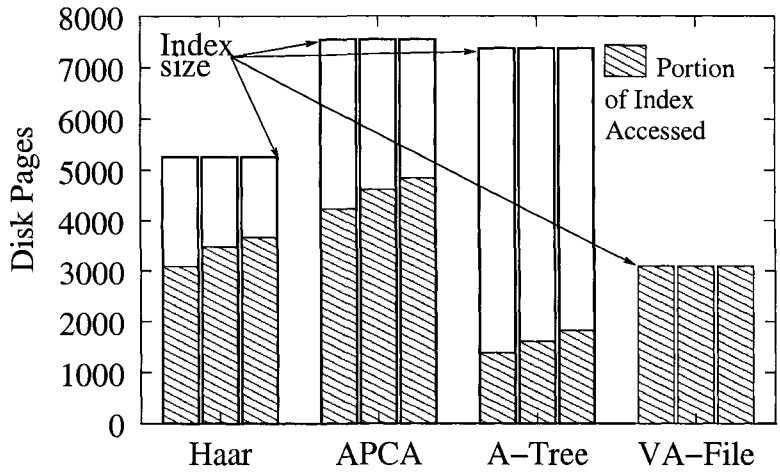


FIGURE 6.2. From Left to Right, Number of Index Pages Accessed for 5, 10, and 20 Nearest Neighbor Search. A Significant Portion of the Indexes is Accessed Indicating Poor Performance

by the small number of *false alarms* it introduced.

Another limitation of most of the techniques surveyed is the use of a hierarchical index for processing similarity search. One drawback of a hierarchical index is the amount of storage overhead it introduces. In the example used in Figure 6.2, the hierarchical indexes produced by the Haar, APCA, and A-Tree methods required 5,250, 7,550, and 7,370 pages of storage, respectively. Considering that the data set requires 24,750 pages of storage, the storage overhead incurred by the indexes used by the Haar, APCA, and A-Tree methods was 21%, 30.5%, and 30%, respectively. In contrast, the storage overhead introduced by the VA-File, which is a linear index, was only 12.5 %. In addition, it is generally accepted that hierarchical indexes are not always effective for similarity search on high-dimensional data such as the approximations of time series. This trend was observed during our own preliminary experiments, as shown in Figures 6.2 and 6.3. The shaded portion of the bars in Figure 6.2 indicates that a large portion of the index was accessed during k-NN search. The bars in Figure 6.3 illustrate the total elapsed time spent on k-NN search, while their shaded portion indicates the amount of CPU time required for the processing

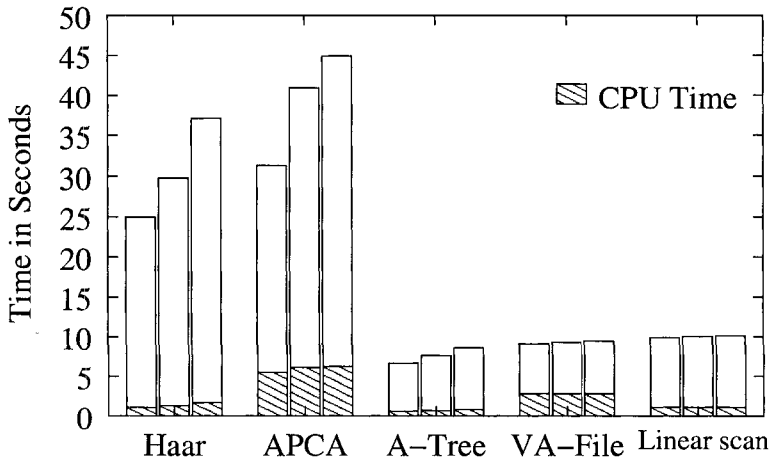


FIGURE 6.3. From Left to Right, Elapsed Time for 5, 10, and 20 Nearest Neighbor Search. Haar and APCA Performed worst than a Simple Linear Scan on the Data, whereas A-Tree and VA-File were not Significantly better

these queries.

The high CPU cost of APCA was caused by a large portion of its index being accessed. For every index node, u, accessed during search, there are u_c distance computations, where u_c is the number of entries in u[1]. In addition, each entry e_i in u is inserted into a priority queue that indicates the order in which index nodes will be expanded. For these reasons, it is clear that accessing a large number of index nodes has a significant impact on the CPU cost of the search.

There are a few lessons to be learned from Figure 6.3. First, simple approximations like those used by the A-Tree and the VA-File can perform better than the more sophisticated approximations provided by the Haar and APCA techniques (however, the performance of the A-Tree and VA-File was not significantly better than the performance of the linear scan). Second, the portion of time spent in computation was only a small fraction of the total elapsed time and most of the time was consumed by IO operations. Hence it is important that we focus our efforts on designing

[1]For the example shown in Figure 6.3, APCA required an average of nearly 65,000 distance computations for each 5-NN query.

data structures and search methods that minimize the amount of time spend on IO operations.

6.2 An IO Efficient Technique for Similarity Search

Our proposal for the efficient processing of similarity search on large time series databases is twofold. First, to reduce the time spend accessing the index (*index search overhead*), we favor the use of a linear index over a hierarchical index. Second, to reduce the *data search overhead*, we propose a new approximation technique that provides tight *upper-* and *lower-bound* distances from a query to data objects. Using both upper- and lower-bound distances allows us to use an r-optimal k-NN search algorithm with a linear index.

In Table 6.1, we define the symbols and notations used in the rest of the chapter.

6.2.1 k-NN Search with a Linear Index

In Section 6.1, we have observed that one significant limitation of hierarchical indexes was the large portion of the index being accessed during search (shaded portion of Figure 6.2). This is because the performance of a hierarchical index rapidly degrades as the number of dimensions increases [12, 125, 126]. In fact, Berchtold *et al.* formally proved that there always exists a dimension d, for which a linear scan on the data is faster than a search based on a hierarchical index [12].

To overcome this problem, we propose the use of linear indexes. Let us provide an example that motivates the use of a linear index over a hierarchical index for k-NN search. We create a linear index using the APCA approximation by organizing the APCA data approximations sequentially in a file following the same order of the data objects. The search algorithm proceeds as follows.

A simple k-NN search algorithm. Sequentially access the approximations in the linear index and compute the corresponding lower-bound distances. We start with an

Notation	Description
O	universe of time series objects
D, $D \subset O$	a data set
N, $N = \|D\|$	number of entries in data set
L, $L \in \mathcal{N}$	length of each time series object
$\overline{x}$, $\overline{x} \in D$	time series data object
$\overline{q}$, $\overline{q} \in O$	time series query object
$\mathcal{A}$	set of SCoBE approximations
$A_s^b \in \mathcal{A}$	an SCoBE approximation
$s \in \mathcal{N}$	segments in the approximation
$e \in \mathcal{N}$	extent of each segment
$b \in \mathcal{N}$	bits per value in approximation
$w \in \Sigma^*, \Sigma = \{0, 1\}$	a bit string
$d : O \times O \rightarrow \mathcal{R}^+$	distance between two objects
$d_{lb} : O \times \mathcal{A} \rightarrow \mathcal{R}^+$	lower-bound distance using the *SCoBE* approximation
$d_{ub} : O \times \mathcal{A} \rightarrow \mathcal{R}^+$	upper-bound distance using the *SCoBE* approximation
$BR : \mathcal{A} \rightarrow O \times O$	bounding region
$\overline{u} \in O$	upper limit of bounding region
$\overline{l} \in O$	lower limit of bounding region

TABLE 6.1. Symbolic Notation Used in this Chapter

empty result set, *result*. As we insert new elements into the result set, we keep them sorted by their distance to the query object, $\overline{q}$. Whenever a lower-bound distance is smaller than the k_{th} largest distance in *result*, we fetch the corresponding time series data object, $\overline{x}$, and compute the distance from $\overline{q}$ to $\overline{x}$. If this distance is still smaller than the largest distance on the result set, we evict from *result* the element with the largest distance to $\overline{q}$ and insert the new entry $\overline{x}$.

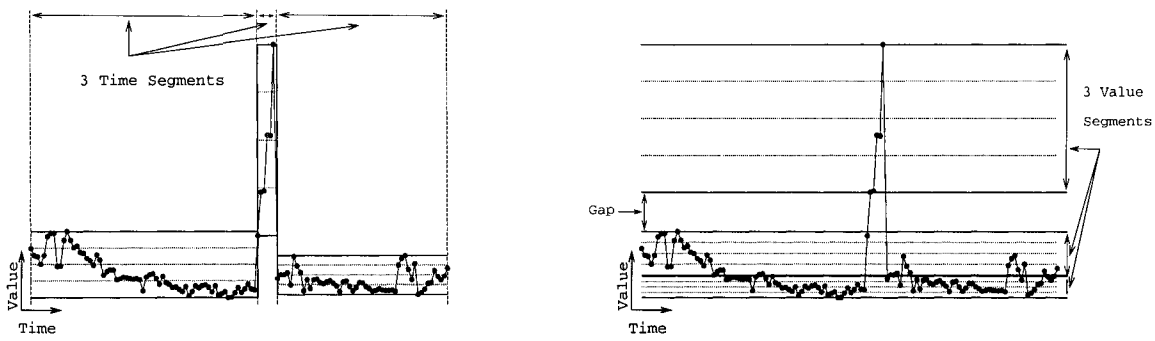
	Tree APCA		**Linear APCA**	
	5-NN	20-NN	5-NN	20-NN
Index pages accessed	4,234	4,843	3,094	3,094
Data objects fetched	7,002	14,129	*24,621*	*25,803*
Elapsed time (sec)	31.26	44.94	*7.08*	*7.39*

TABLE 6.2. Performance Gains from Switching to a Linear Index

In Table 6.2, we present three performance metrics for nearest neighbor search using the APCA approximation. We organized the APCA approximations using an R-Tree (*Tree APCA*) and a linear index (*Linear APCA*). Despite the larger number of disk pages read by *Linear APCA*, the overall performance was better than *Tree APCA*. The performance gains observed on *Linear APCA* are due to the IO access pattern of this technique. For *Tree APCA* every page is randomly accessed. The linear index of *Linear APCA*, on the other hand, is read sequentially. Similarly, data objects are fetched by *Linear APCA* in increasing order by their position on the data file.

6.2.2 A new Representation for Time Series: *SCoBE*

In this section, we present a new data representation for time series, *Self COntained Bit Encoding (SCoBE)*. *SCoBE* approximates a time series data by first segmenting it into disjoint subsequences. The segmentation can be performed either along the *time* dimension or along the *value* dimension. An example of segmentation is shown in Figure 6.4. Figure 6.4(a) illustrates *time* segmentation, whereas Figure 6.4(b)



(a) Time Segmentation (b) Value Segmentation

FIGURE 6.4. Time and Value Segmentation for *SCoBE*

illustrates *value* segmentation. Note that there is a gap in Figure 6.4(b). Since no values in the time series data fall within the value range of the gap, the segments defined by the approximation do not need to cover it. In both cases three segments are generated. For each segment, we record the minimum and maximum observed values. This provides the range of possible values within the segment. Then, each segment's value range is partitioned into 2^b cells, where b is a user defined number of bits (these partitions are illustrated by the horizontal dotted lines on each segment in Figure 6.4, where $b = 2$).

For example, consider a segment $\overline{s_i}$ of a time series object, $\overline{s_i} = <1.0, 1.5, 1.8, 1.1, 2.4, 3.7, 4.5, 5.0, 3.6, 4.8>$. The range of values for this segment is $[1.0, 5.0]$. If we use 2 bits per value to encode this segment, we can partition the value range into 4 cells, namely $c_0 = [1.0, 2.0)$, $c_1 = [2.0, 3.0)$, $c_2 = [3.0, 4.0)$, and $c_3 = [4.0, 5.0]$. We assign a bit encoding for each cell: 00 for c_0, 01 for c_1, 10 for c_2, and 11 for c_3. Since each value in the segment is represented by the bit encoding of the cell they fall into, the encoding for the values in segment $\overline{s_i}$ will be the string $w =$ 00 00 00 00 01 10 11 11 10 11.

We now provide details on how the segmentation and encoding of values is performed by *SCoBE*.

Segmenting Time Series on the Time Dimension: There are two alternatives for segmenting a time series along the time dimension. Either fixed- or variable-length segments can be used. In the former case, it is sufficient to know the number of segments to determine the length as well as the values covered by each segment. In the latter, we need to explicitly maintain the length of each segment.

For fixed-length segmentation, we partition a time series of length L into s segments of length e. We assume $L = s \times e$. Otherwise, the length of the last segment is simply (L *mod* e). If variable-length segments are used, the length of each segment is chosen based on a cost estimate. In particular, the goal is to optimize a function (defined in Section 6.2.3) that measures the quality of the resulting approximation. Once the segments have been defined, we quantize the values in each segment with respect to its range. For this, we partition the range into 2^b cells, where b is a user defined number of bits, and assign a bit encoding to each of these cells. Each value in the segment is then approximated by the bit string (*i.e.*, between 0 and $2^b - 1$) representing the cell where the value falls.

Formally, each time-segmented *SCoBE* approximation is a sequence of s quadruples of the form $<min_i, max_i, e_i, w_i>$ defined as follows.

Definition 7 (Time-segmented SCoBE). *Given a time series* $\overline{x}$ *of length* L, $\overline{x} = <x_1, \ldots, x_L>$ *partitioned into* s *segments, we define the Self COntained Bit Encoding approximation* A_b^s *of* $\overline{x}$ *as*

$$A_b^s(\overline{x}) = <T_1, \ldots, T_s>,\ T_i = <min_i, max_i, e_i, w_i>,$$

where for each segment $\overline{s_i}$ *of length* e_i, $min_i = min(\{x_j \mid x_j \in \overline{s_i}\})$, $max_i = max(\{x_j \mid x_j \in \overline{s_i}\})$, *and* w_i *is the concatenation of the* b*-bit encoding of all* x_j, $\forall x_j \in \overline{s_i}$. *Note that, if fixed-length segments are used, there is no need to store the length of each individual segment and* e_i *is omitted.*

The algorithm to compute $A_b^s(\overline{x})$ using fixed-length segments is quite simple and

the *SCoBE* approximation can be generated in $\mathcal{O}(L)$ time. If we use use variable-length segments, the algorithm's complexity is no longer linear. An optimal solution can be generated using dynamic programming in $\mathcal{O}(L^3 \times s)$ time. Instead, we opted for a greedy algorithm that finds a sub-optimal solution in $\mathcal{O}(L^2)$ time (L is the length of $\bar{x}$, not the size of the data set). This algorithm iteratively merges the two adjacent segments that increase the cost estimate (defined in Section 6.2.3) by the least amount.

Segmenting Time Series on the Value Dimension: Segmenting a time series along the *value* dimension is slightly different and more complicated than the conventional time segmentation. In part, this is due to the fact that values in a particular range do not necessarily correspond to a consecutive subsequence in the time series. In this case, not only do we need to keep track of the ranges and extents of the different segments but also we need to maintain the mapping between segments and values.

To find an optimal segmentation along the *value* dimension, we start by sorting values in a data time series, $\overline{x}$, in increasing order. Let us denote the list of sorted values by $\overline{x'}$. Now, we are looking for the s segments of $\overline{x'}$ that minimize our cost estimate. Note that the optimal segmentation problem is quite similar to finding optimal histograms (*i.e.*, our segments can be seen as the histogram's bins). For this, a dynamic programming algorithm exists such that an optimal solution can be computed in $\mathcal{O}(L^2 \times s)$ time [64], provided that the cost function on each segment can be computed in constant time. Since our cost estimate depends on the *min*, *max*, and *count* of each segment (as shown by Equation 6.2 in Section 6.2.3), the cost of each segment can be computed in constant time. This is possible because $\overline{x'}$ is sorted by values, which allows us to have access to the *min* and *max* values of a segment in constant time. Note that we cannot have access to the *min* and *max* values of a segment in constant time if we only use $\overline{x}$ (*e.g.*, as in the case of time segmentation), in which case the optimal solution requires $\mathcal{O}(L^3 \times s)$ time.

After the segments have been determined, we discard $\overline{x'}$ and use only $\overline{x}$. Each segment is partitioned into 2^b cells and each value is encoded using the bit string corresponding to the segment and partition where the value falls. Therefore, each value is mapped to a bit string of size $log_2(s) + b$, where the first $log_2(s)$ bits identify the segment containing the value and the last b bits identify the partition within the segment.

Formally, the value-segmented *SCoBE* approximation of a time series $\overline{x}$, using s segments is a pair $< SegmentList, w >$ defined as follows.

Definition 8 (Value-segmented SCoBE). *Given a time series $\overline{x}$ of length L, $\overline{x} = < x_1, \ldots, x_L >$ partitioned into s segments, we define the Self COntained Bit Encoding approximation A_b^s of $\overline{x}$ as*

$$A_b^s(\overline{x}) = < \{R_1, \ldots, R_s\}, w >, \; R_i = < min_i, max_i >,$$

where $min_i = min(\{x_j \mid x_j \in \overline{s_i}\})$, $max_i = max(\{x_j \mid x_j \in \overline{s_i}\})$ for each segment $\overline{s_i}$, and w is the concatenation of the $(log_2(s) + b)$-bit encoding of all the values in $\overline{x}$.

6.2.3 Cost Function and Quality of SCoBE

In this section we define a measure of the quality of the proposed approximation. Given our interest in improving search performance by reducing the number of false alarms, the cost estimate we define is based on the probability of a data approximation being intersected by a query search radius. Our focus is then to minimize this probability.

SCoBE represents a time series data by a sequence of cells that define a region bounding the original data object. The Bounding Region (BR) created by $A_b^s(\overline{x})$ is a pair of upper- and lower-bounding time series, $\overline{u}$ and $\overline{l}$. Formally,

$$BR = < \overline{u}, \overline{l} >, \; \overline{u} = < u_1, \ldots, u_L >, \text{ and } \overline{l} = < l_1, \ldots, l_L >$$

The values of $\bar{u}$ and $\bar{l}$ are generated from $A_b^s(\bar{x})$ as follows.

$$u_i = min_j + r \times (w_i + 1), \quad l_i = min_j + r \times w_i,$$
$$r = (max_j - min_j)/2^b$$

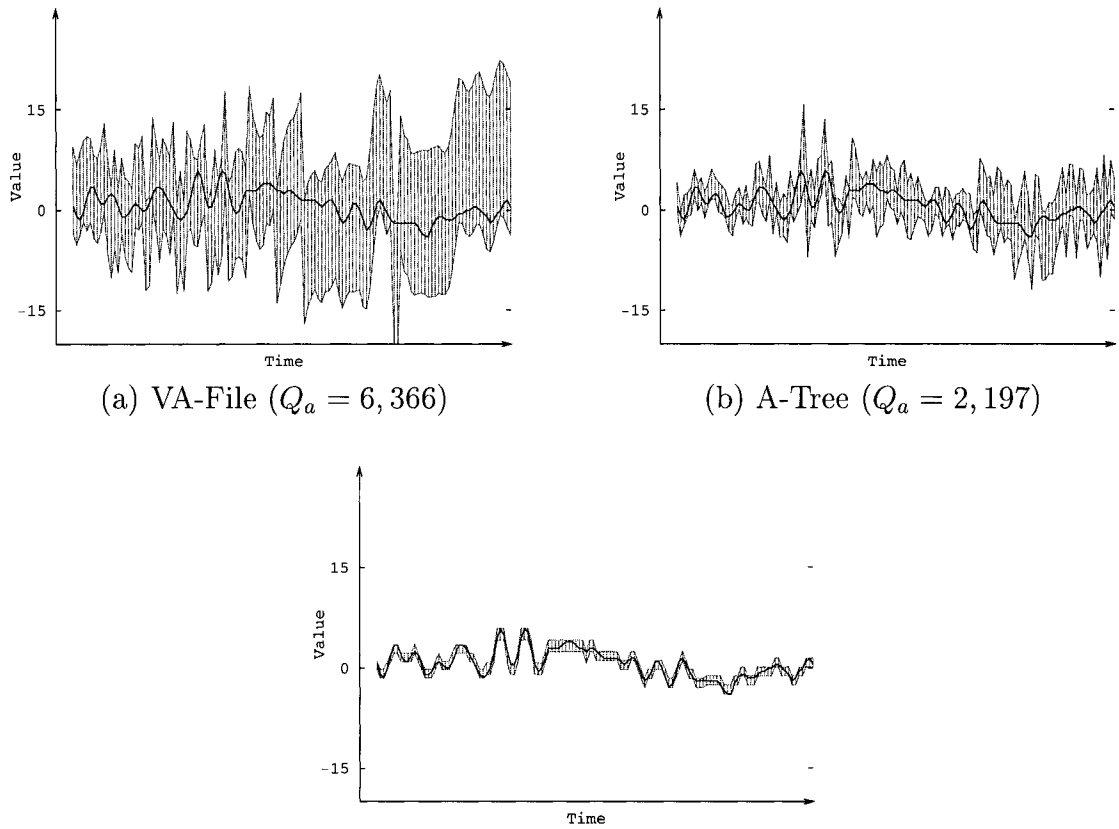
where $1 \leq i \leq L$, min_j and max_j define the range of values for the j_{th} segment in $\bar{x}$, and such segment contains the i_{th} value in $\bar{x}$. Finally, w_i is the b-bit encoding of the i_{th} value in $\bar{x}$. In our previous example, where the segment $\overline{s_i}$ of a time series object was encoded using 2 bits per value, $\overline{s_i} = <1.0, 1.5, 1.8, 1.1, 2.4, 3.7, 4.5, 5.0, 3.6, 4.8>$, the resulting values for $\bar{u}$ and $\bar{l}$ are $<2, 2, 2, 2, 3, 4, 5, 5, 4, 5>$ and $<1, 1, 1, 1, 2, 3, 4, 4, 3, 4>$, respectively.

SCoBE defines upper- and lower-bound distances with respect to this bounding region (Section 6.2.5). Furthermore, the quality of these distances depends on the size of the region defined by the approximation. For instance, if the size of the region is minimal (*i.e.*, its area is zero), then the values of the upper- and lower-bound distance will exactly match the value of the actual distance between the query and data time series. As we increase the area of the bounding region, the difference between the values of the upper- and lower-bound distance and the value of the actual distance also increases. In other words, a smaller area yields tighter upper- and lower-bound distances, which reduces the number of false alarms during search. Therefore, we can estimate the quality of the *SCoBE* approximation using the area of the region that it defines. Alternatively, we could think of this bounding region as the error introduced by the approximation. Hence, by minimizing the area of this region, we minimize the amount of introduced error.

We formally define our cost estimate, Q_a, the quality of the approximation, as

$$Q_a(A_b^s(\bar{x})) = area(BR) = \sum_{i=1}^{L}(u_i - l_i), \tag{6.1}$$

where $BR = <\bar{u}, \bar{l}>$, L is the length of the time series data object, and $u_i \in \bar{u}$, $l_i \in \bar{l}$, for $1 \leq i \leq L$.



(a) VA-File ($Q_a = 6,366$)

(b) A-Tree ($Q_a = 2,197$)

(c) SCoBE ($Q_a = 371$)

FIGURE 6.5. VA-File, A-Tree, and *SCoBE* Approximations and Their Quality

In particular, for the *SCoBE* approximation we can efficiently compute Q_a using the segments defined by the approximation. Because all cells within a segment are the same size, it is possible to define Q_a as

$$Q_a(A_b^s(\bar{x})) = \sum_{i=1}^{s}(count_i \times \frac{(max_i - min_i)}{2^b}), \tag{6.2}$$

where $count_i$ is the number of values contained by segment i, whose range is defined by max_i and min_i, and 2^b is the number of partitions within a segment.

We can define Q_a for any approximation that generates a bounding region and uses it for computing upper- and lower-bound distances. In Figure 6.5, we show an example of a time series and its corresponding approximation by the VA-File, the A-

Tree[2], and the *SCoBE* methods. Each approximation reduces a time series data to $\frac{1}{8}$ of its original size. This figure clearly shows the superiority of *SCoBE* for approximating time series data. While the APCA approximation also defines a bounding region, its bounding region is only used for index construction and not during the search process for the computation of lower-bound distances. Because of this, we did not include the APCA approximation in Figure 6.5.

6.2.4 Building an Adaptive Linear Index

In Section 6.2.2, we have defined different methods for computing *SCoBE*. Such methods are based on either time or value segmentation. In general, given a constraint on the size of the approximation, it is not possible to know in advance which method will produce the best approximation. However, we have the advantage of a cost estimate that can be evaluated on an individual basis. That is, for each time series data object, we generate *SCoBE* approximations using all methods described in Section 6.2.2 and compute their quality using Equation 6.2. After this, we simply select the best approximation for the given time series. Our proposal is to build the linear index using the *SCoBE* approximation with the minimum cost. This implies annotating each approximation with a tag indicating the method used for its generation. Since only a small number of methods is used, this annotation only adds a couple of bits to the approximation.

6.2.5 Upper- and Lower-Bound Distances

Since the *SCoBE* approximation defines a region bounding a data object, it is easy to define both upper- and lower-bound distances for any L_p norm. Given a query time series $\overline{q}$ and a data object $\overline{x}$, both of length L, and the bounding region BR

[2]For the A-Tree, the approximation of a time series data object is defined with respect to its enclosing MBR (*i.e.*, its parent node).

corresponding to the *SCoBE* approximation of $\overline{x}$, $A_b^s(\overline{x})$, we define a lower-bound distance $d_{lb}(\overline{q}, A_b^s(\overline{x}))$ and an upper-bound distance $d_{ub}(\overline{q}, A_b^s(\overline{x}))$ as follows.

Definition 9 (Lower-bound distance).

$$d_{lb}(\overline{q}, A_b^s(\overline{x})) = \sqrt[p]{\sum_{i=1}^{L} |d_{min}(q_i, u_i, l_i)|^p} \tag{6.3}$$

$$d_{min}(q_i, u_i, l_i) = \begin{cases} q_i - u_i & \text{if } q_i > u_i \\ l_i - q_i & \text{if } q_i < l_i \\ 0 & \text{otherwise.} \end{cases}$$

Definition 10 (Upper-bound distance).

$$d_{ub}(\overline{q}, A_b^s(\overline{x})) = \sqrt[p]{\sum_{i=1}^{L} |d_{max}(q_i, u_i, l_i)|^p} \tag{6.4}$$

$$d_{max}(q_i, u_i, l_i) = max(|q_i - l_i|, |u_i - q_i|)$$

Lemma 4. *Given a query object $\bar{q}$, a time series data object $\bar{x}$, and its SCoBE approximation, $A_b^s(\bar{x})$, the following condition holds.*

$$d_{lb}(\bar{q}, A_b^s(\bar{x})) \leq \sqrt[p]{\sum_{i=1}^{L} |q_i - x_i|^p} \leq d_{ub}(\bar{q}, A_b^s(\bar{x}))$$

A lower-bound distance guarantees no *false dismissals* when evaluating a similarity search query [42]. An upper-bound distance, on the other hand, is not necessary for finding correct search results. However, it can be used to further reduce the search space.

6.2.6 Quantizing Long Sequences

In Section 6.2.2 we described how to obtain the *SCoBE* representation of a time series using either time or value segmentation. Because our algorithms have a quadratic time complexity on the length of the time series, the performance will suffer when

trying to generate the *SCoBE* representation of long time series. To overcome this minor drawback, long sequences can be processed in small pieces (*i.e.*, small disjoint subsequences). After this, the results can just concatenated to obtain the *SCoBE* representation of the entire sequence. Because concatenation can be performed in constant time, working with small sequences significantly reduces the processing time for quantizing long sequences.

6.3 Similarity Search on a Linear Index

In this section, we present an algorithm for similarity search on time series data. We first describe how to processes *whole sequence match* queries using *SCoBE*. Then, we show how *SCoBE* can be used for evaluating two other types of similarity search queries for time series data, namely *dynamic time warping* and *subsequence match*.

6.3.1 A Basic Algorithm for k-NN Whole Sequence Match Using Upper-bounding Distances

Weber *et al.* proposed a two-stage algorithm for nearest neighbor search based on the VA-File [126] (see Figure 6.6). Weber's algorithm is a variant of the optimal multi-step k-NN algorithm proposed by Seidl and Kriegel [110]. The first stage of this algorithm (**filtering stage**) uses both upper and lower-bound distances to discriminate between irrelevant entries and candidates to the answer. During this stage, the entire linear index is scanned. The k_{th} smallest upper-bound distance, d_k^u, found so far is maintained. If the lower-bound of an approximation is larger than d_k^u, the corresponding object is simply ignored. Otherwise, the object is possibly part of the solution and is inserted into a *candidate set*.

In the second stage (**data access stage**), the elements in the *candidate set* are visited in increasing order by their lower-bound distance. For each such candidate, the corresponding data object is fetched from the data set and the distance to the query

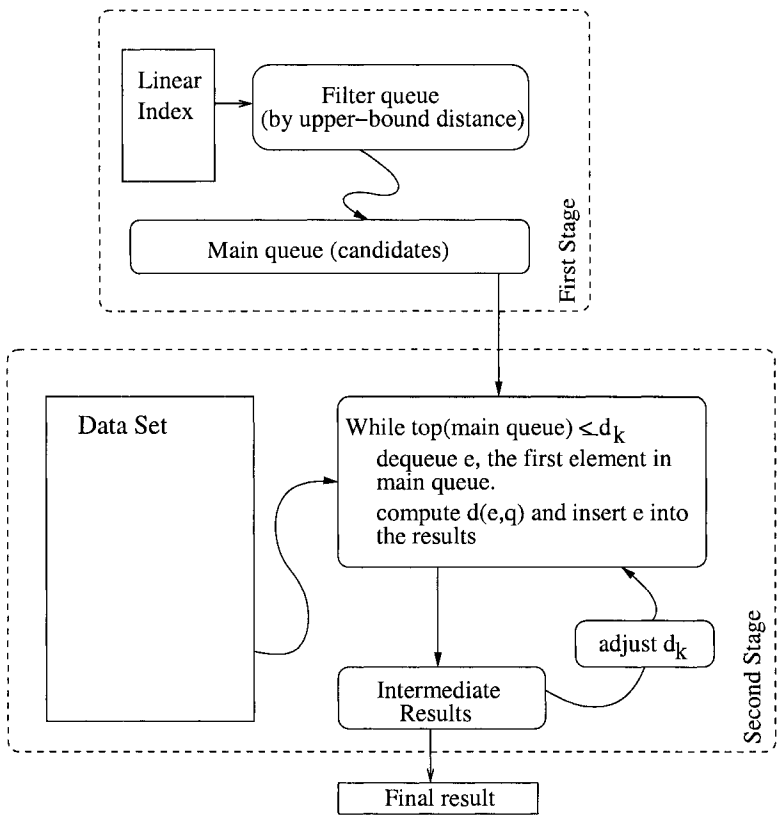


FIGURE 6.6. Illustration of k-NN Search on a Linear Index

object $\bar{q}$ is computed. This process stops when the smallest lower-bound distance in the remaining *candidate set* is larger than the current k_{th} nearest neighbor distance. This termination condition guarantees that only data corresponding to candidates with a lower-bound distance smaller than the k_{th} nearest neighbor distance is fetched. Therefore, this algorithm is r-optimal as defined by Seidl and Kriegel [110].

6.3.2 An Improved k-NN Search Algorithm for Whole Sequence Match

The overall performance of Weber's algorithm can be improved by adding a third stage in the search process. We propose to pre-process the *candidate set* before entering the data access stage. The idea is to identify objects in the *candidate set* that are positively part of the answer. This should be done at a minimal computational cost

and without adding IO operations. To support our idea, let us present the following lemma.

Lemma 5. *When searching for k-nearest neighbors, for every element, o_i, in the* candidate set, *with upper-bound distance smaller than the $(k+1)_{th}$ nearest neighbor distance, the corresponding data object, $\bar{x}_i$, is among the k nearest neighbors.*

Proof. Let o_i denote an element of the *candidate set* and $\bar{x}_i$ denote o_i's corresponding data object. Furthermore, assume $o_i.lb$ and $o_i.up$ are lower- and upper-bounds to $d(\bar{q}, \bar{x}_i)$, respectively, where $\bar{q}$ is a query object. Since, $o_i.lb \leq d(\bar{q}, \bar{x}_i) \leq o_i.up$, and $o_i.up < d_{k+1}$, it follows that $d(\bar{q}, \bar{x}_i) < d_{k+1}$. Therefore, $\bar{x}_i$ is part of the answer. □

The direct application of lemma 5 to Weber's algorithm is not possible because the value of d_{k+1} is only known after the search process has completed. Instead, we estimate d_{k+1} by the $(k+1)_{th}$ smallest lower-bound distance in the *candidate set.* Let $\hat{d}_{k+1}$ denote the estimated value of d_{k+1}. Because $\hat{d}_{k+1} \leq d_{k+1}$, using $\hat{d}_{k+1}$ instead of d_{k+1} can result in a smaller number of objects satisfying lemma 5. However, this will not result in any *false dismissals.*

We illustrate this pre-processing stage in Figure 6.7. Let us represent each element in the *candidate set* by a segment $[o_i.lb, o_i.ub]$ defined in a one-dimensional *distance* space. The bold point on each segment indicates the distance from the corresponding data object to the query object. The vertical dotted line at $\hat{d}_{k+1}$, indicates the estimation of d_{k+1}. In this example, k is 5. In Figure 6.7, segments a, b, c, and d are all to the left of $\hat{d}_6$. In consequence, we can accept their corresponding data objects as part of the solution without further processing. Segments e and f, on the other hand, intersect $\hat{d}_6$. For them, we cannot determine whether they should be added to the result set without fetching their corresponding data objects.

The idea of a three-staged search is shown in Algorithm 5. Between the two stages of Weber's algorithm, a new stage is added to identify objects that are part of the

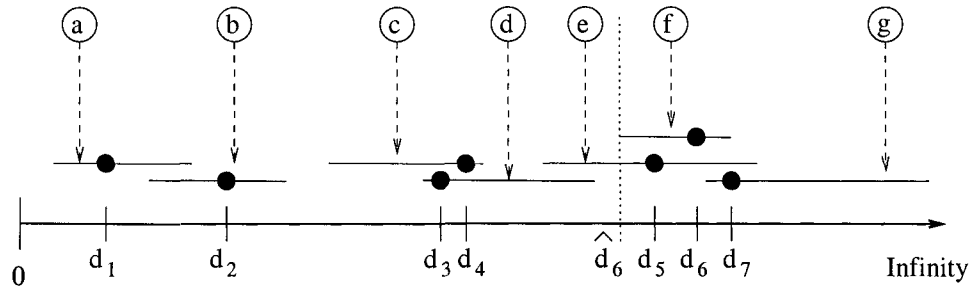


FIGURE 6.7. Early Identification of Entries in the Answer. There is no Need to Access Data for Objects a, b, c, and d

answer without fetching additional data. The filtering stage is contained by lines 1 to 5. To efficiently keep track of the k_{th} smallest upper-bound, we use *filter*, a priority queue maintained in descending order and limited to only k elements. The candidate set is implemented as a priority queue, *main*, as well. In this case, the priority is given by the lower-bound distance and the queue is maintained in ascending order.

The proposed optimization stage, is defined by lines 6 to 11 in Algorithm 5. We use a priority queue to store the result set, *result*, kept in descending order. In line 9, we check if an entry in the candidate set (*i.e.*, *main* queue) belongs to the answer. Objects identified as part of the answer are inserted in *result* with priority 0. Otherwise, the corresponding data is fetched and the distance, *dist*, to the query object is computed. In this case, objects are inserted in *result* with priority *dist*.

During the data access stage, we dequeue candidates from the main queue, fetch the corresponding data object and compute the distance, *dist*, from the query to the data object. If *dist* is smaller than the current k_{th} nearest neighbor distance, d_k, the element in the result set with the largest distance is evicted and the new object inserted. This process stops when the smaller lower-bound in *main* is larger than the largest distance in *result*. Note that the second stage of Algorithm 5 does not fetch any data object that would have not been fetched by the data access stage of Weber's algorithm. Hence, Algorithm 5 preserves the r-optimality.

Note that without a good upper-bound distance, Algorithm 5 would not filter

Algorithm 5: Improved k-NN search on a linear index.

```
   Input: q̄, k        Output: result
   // q̄ is the query object.
   // k is the number of nearest neighbors searching for.

   // first stage: filtering out irrelevant entries.
 1 for each approximation, A, in the list do
 2     lbound ← d_lb(q̄, A); ubound ← d_ub(q̄, A);
 3     if (|filter| = k) ∧ (ubound < top(filter).prio) then
           dequeue(filter);
       end
 4     enqueue(filter, A.id, ubound);
 5     if (|main| < k) ∨ (lbound < top(filter).prio) then
           enqueue(main, A.id, lbound, ubound);
       end
   end
   // second stage: index-based identification of objects in answer.
 6 d̂_{k+1} ← the lbound of the (k + 1)_th entry in main;
 7 for i = 1 to k do
 8     o ← dequeue(main);
 9     if o.ubound < d̂_{k+1} then enqueue(result, o.id, 0);
       else
10         x̄ ← fetch(o.id); dist ← d(q̄, x̄);
11         enqueue(result, o.id, dist);
       end
   end
   // third stage: data access.
12 d_k ← top(result).prio;
13 while (main is not empty) ∧ (top(main).prio ≤ d_k) do
14     o ← dequeue(main);
15     x̄ ← fetch(o.id); dist ← d(q̄, x̄);
16     if dist < d_k then
17         if |result| = k then dequeue(result);
18         enqueue(result, o.id, dist);
19         d_k ← top(result).prio;
       end
   end
```

out a significant number of irrelevant entries. In consequence, the first stage could degenerate into sorting all entries in the database by their lower-bound distance to the query object. Given the non-trivial complexity of sorting, and the large number of entries in a database, the tight upper-bound distance provided by *SCoBE* plays an important role in reducing memory and computation requirements during similarity search.

6.3.3 Dynamic Time Warping (DTW) Queries

It has been shown that, if we limit the warping path, we can use an index-based approach to answer dynamic time warping queries [69]. Essentially, a constrained warping path creates an envelope around the query time series. This envelope can be approximated using a modified Piecewise Aggregate Approximation (PAA), which fully contains every point in the envelope (*i.e.*, using the `MIN` and `MAX` aggregate functions instead of `AVG`). This approximation to the envelope can then be used to define a distance function that lower-bounds the dynamic time warping distance. The concept of envelop was later improved by Zhu and Shasha [137]. They proved that the approximation to the envelope does not need to contain every point in the envelope but only every approximated point in the envelope. This property is termed *container-invariant*. A container-invariant transformation of an envelope guarantees no false dismissals during dynamic time warping queries [137].

To evaluate DTW queries using *SCoBE*, we simply define an envelope around the query time series by limiting the warping path. The warping path can be limited using the Itakura parallelogram or the Sakoe-Chiba band [99, 107], for instance. We then use Algorithm 5 but substitute the query time series by its envelope when computing the upper- and lower-bound distances (line 2). In addition, we use DTW distance during the data access step of the algorithm (line 15). Note that, when using *SCoBE*, we do not need to apply any other transformation to the query envelope.

6.3.4 Subsequence Match Queries

We have revisited this problem in the light of technological advances during the last ten years. In particular, we have paid close attention to the increasing ratio of CPU to disk performance. We conducted a series of benchmarks using slow computers (for today's standards) and fast disks. Our findings show that it is possible to compute Euclidean distance from a query to all subsequences contained in a disk page faster than the average time required for fetching a random disk page. These results indicate that, if the number of candidate subsequences for a particular query is larger than the number of disk pages containing the data it will be faster to process this query using linear scan than using an index-based approach. Therefore, finding a method that drastically reduces the number of candidates during subsequence match queries is highly desirable.

Based on these findings, we propose to use *SCoBE* for subsequence match. A naïve approach is presented first. While it works, the computational cost is too high as the distance to every subsequence needs to be computed. This problem is minimized in a second approach by the use of moving averages. However, storing moving averages for every time point in the time series has a negative effect on the size of the data representation. Our final approach consists on quantizing the moving averages along with the values in the time series.

Brute Force Approach: A very simple approach for evaluating subsequence match queries using *SCoBE* is to create a linear index in the same way as we do for processing whole sequence match queries. Logically, this linear index contains the *SCoBE* approximation for all possible subsequences of length w (*i.e.*, the size of the query). Therefore, it is possible to apply Algorithm 5 on this linear index to answer subsequence match queries. For an efficient implementation, however, instead of decoding every possible subsequence to compute upper- and lower-bound distances to the query (first step of Algorithm 5), we decode the approximation for a data time series just

once and then compute the upper- and lower-bound distances to all subsequences by sliding the query over the decoded data approximation. After this minor adaptation, we use Algorithm 5 to evaluate subsequence match queries.

Using Moving Averages: One major drawback of the brute force approach is its computational cost. We can reduce this cost if we are capable of filtering out subsequences using a less expensive function instead of computing the distance to every single subsequence in the time series. To reduce the computational cost, we propose to use moving averages with a window of length l for every value in the time series, where $w = k \times l$ and w is the length of the query sequence. At the same time, the query sequence is partitioned into k disjoint segments of length l. For every segment, we compute the average of its values. Using the moving averages of the data time series and the segmented means of the query sequence, we can find a lower bound to any L_p norm distance to the query sequence (see [131]).

To evaluate a subsequence match query, we use the moving averages and *SCoBE* in tandem. That is, we compute a lower-bound to the distance between the query and a data subsequence using the moving averages. If this distance is better than the current best candidate, we compute a lower-bound on the distance using *SCoBE*. Because the lower-bound distance based on moving averages can filter out a great number of candidates, the computational costs of evaluating a subsequence match queries are reduced almost by a factor of k with respect to the brute force approach. However, because we add one extra value for every time point in the data time series, the storage overhead of this approach is significant. In fact the linear index used during the search is larger than the data itself.

Using Quantized Moving Averages: The previous two approaches for evaluating subsequence match queries using *SCoBE* had drawbacks due either to computational overhead or to storage overhead. We overcome these two drawbacks by finding a compromise between the computational and the storage overhead. In particular, we

propose to quantize the values of the moving averages using *SCoBE*. By doing so, we reduce the storage requirements for the linear index. Unfortunately, this has a negative effect on the the quality of the lower-bound distance provided by the moving averages because we lose precision by quantizing the values. Note that this degradation on the lower-bound distance provided by the moving averages only affects the number of irrelevant candidates it can prune away. It has no effect in the correctness.

To evaluate a subsequence match query, we proceed in the same way as we did for the previous approach, except that now we need an extra step where we decode the value of the quantized moving averages. After this, a lower-bound distance can be defined using the decoded moving averages. Therefore we can filter out irrelevant candidates. If a candidate cannot be filtered out using the moving averages, *SCoBE* is used to compute a tighter lower-bound distance.

6.4 Performance Evaluation

In this section, we empirically demonstrate the performance benefits of using *SCoBE* organized in a linear index during exact k-NN search. The main focus of this section is the evaluation of whole sequence match queries. However, we also include experiments for subsequence match queries. Unless we indicate otherwise, k-NN search refers to whole sequence match. In our experiments, we used 5 data sets from different sources. We compared our proposed technique, *SCoBE*, to state of the art methods for similarity search on time series data. Three different performance metrics were used to evaluate the performance of the different techniques presented in this experimental study.

6.4.1 Data Sets

Our test data sets were obtained from sources of different fields such as medicine, finance, and astrophysics. We did this in an effort to minimize *data bias*[3] in our experiments. Four data sets were from real world data sources and one was synthetically generated. Just as we did in Chapter 5, from each data source we generated 3 data sets each of which contained time series objects of length 256, 512, or 1024. Each data set contains 100,000 time series objects of the same length. We randomly extracted 1,000 entries from each data set. This subset of 1,000 time series objects became our query set. We did this to avoid exact matches with queries in our experiments. In addition, this practice allowed us to use a query object that is in the same domain as the data set. Details of our testing data are given next.

(1) *Plasma:* We obtained this data from the Coordinated Heliospheric Observations Web server (COHOWeb) [33]. COHOWeb provides access to hourly resolution magnetic field and plasma data from 12 different heliospheric spacecraft. In our experiments, we used the plasma temperatures reported by these spacecraft since 1963.

(2) *EEG:* This data arose from a large study to examine electroencephalogram correlations of genetic predisposition to alcoholism. It contains measurements from 64 electrodes placed on subject's scalps which were sampled at 256 Hz (3.9-msec epoch) for 1 second [61].

(3) *ECG:* This is the same ECG data set described in Chapter 5.

(4) *Mixed Bag:* This is the same Mixed Bag data set described in Chapter 5.

(5) *FinTime:* This is the same FinTime data set described in Chapter 5.

[3] *Data bias* is the conscious or unconscious use of a particular set of testing data to confirm a desired finding [71].

6.4.2 Performance Metrics

In our experiments, we evaluated the efficiency of different techniques using four metrics. We measured the storage requirements of the index (*index to data ratio*). In addition, we measured *index search overhead* and the number of *data objects fetched* as the two main factors affecting overall performance of similarity search. Finally, we measured *elapsed time* as the performance metric directly perceived by the user.

(1) *Index to data ratio* is the storage requirement of an index measured as the percentage of the size of the data set. We computed the *index to data ratio* as

$$(SizeOfIndex/SizeOfData) \times 100.$$

(2) *Index search overhead* is the time, in seconds, spent on accessing an index during k-NN search. We estimated this time by

$$\frac{\#ofIndexPagesAccessed}{Total\#ofPagesAccessed} \times (Elapsedtime - CPUtime).$$

(3) *Data objects fetched:* It has been suggested that we should prevent performance bias due to disparities in the quality of implementation of the methods being compared [71]. Following this recommendation, we evaluate the effectiveness of different k-NN search methods by the *number of data objects fetched* from the database. The number of data objects fetched during a k-NN query represents a performance metric that is independent of the quality of implementation.

(4) *Elapsed time:* We used wall-clock time to measure the elapsed time during the evaluation of k-NN queries. This time includes both CPU and IO time. In addition, for each query we recorded the time spent on CPU operations only. This allows us to estimate the time spent in IO operations. Finally, to avoid any caching effects on the index and data files, we flushed the main memory between consecutive queries by loading irrelevant data into the system.

For each metric used in our experiments, we present the average value after executing 100 k-NN queries.

6.4.3 Evaluated Techniques

To show the performance advantages of our proposed technique, we compared it to some of the widely accepted techniques for k-NN search for time series. We also compared *SCoBE* to techniques used for high-dimensional data but not specifically designed for time series data. For completeness, we included linear scanning of the data set in our experiments.

(1) *Linear scan:* We performed a simple linear scan on all the time series objects in the data set to determine the exact k-nearest neighbors of the query.

(2) *VA-File:* We generated the Vector Approximation File for each data set. We used Algorithm 5 to search for the exact k-nearest neighbors of the query. We have excluded the VA$^+$-File [45] from our experiments because the use of the KLT transform is less than ideal for database applications.

(3) *Haar Discrete Wavelet Transformation (Haar):* We computed the Haar discrete wavelet transformation for all the objects in the data set. We used the first c_h coefficients of the Haar transformations and indexed the resulting vectors of c_h coefficients using an R-tree index. We used the GiST library to implement the R-Tree index [60] and implemented Seidl's optimal multi-step k-NN algorithm [110] to search for the exact k-nearest neighbors of the query.

(4) *A-Tree:* We use the A-Tree as a representative of the family of methods combining a hierarchical index and quantization (*e.g.*, A-Tree [108] and IQ-Tree [11]). We built the A-Tree as described by Sakurai *et al.* [108]. We used Seidl's optimal multi-step k-NN algorithm [110] to search for the exact k-nearest neighbors of the query.

(5) *Adaptive Piecewise Constant Approximation (APCA):* We computed the APCA representation for all the objects in the data set. We used c_a segments to represent time series data. The resulting vectors of $2 \times c_a$ coefficients were indexed using both the Hybrid-Tree [20] and the R-Tree index. We compared the Hybrid-Tree and R-tree implementations and found no significant difference in the number of index pages accessed and data objects fetched. In Keogh *et al.* performance evaluation [74], Hybrid-Tree indexes were built in memory and were used to simulate index IO operations by counting the number of accessed tree nodes. Since we were interested in measuring the elapsed time including the time spent during IO operations, we opted for a disk-based index and used the GiST library to implement the R-Tree [60]. Finally, we implemented the same search algorithm described by Keogh *et al.* [74] to find the exact k-nearest neighbors of the query.

(6) *Self COntained Bit Encoding (SCoBE):* We generated a linear index using the *SCoBE* approximation as described in Section 6.2.2. We used Algorithm 5 to search for the exact k-nearest neighbors of the query.

We used the L_2 norm as our similarity metric in the experiments. All indexes were built using pages of 8 KBytes, except for the A-Tree. The A-Tree requires to store an MBR using the original length, L, of the time series objects in the data set. For this reason, and due to the long time series in our data sets, the A-Tree was built using 8-KByte pages for data sets with time series objects of length 256, 16-KByte pages for objects of length 512, and 32-KByte pages for objects of length 1024. We used 32-bit words for storing the values of time series data. The experiments were performed on Intel Pentium computers with 600 MHz processors and Linux operating system. Each computer has 128 MBytes of main memory and 9 GBytes of disk storage connected with a SCSI interface.

For the Haar and APCA transformations, we built indexes using feature vectors of 8, 16, 32, and 64 dimensions (*i.e.*, the values of c_h and c_a). There is a performance trade-off depending on the size of the feature vector. On one hand, a higher-dimensional vector provides a more accurate representation reducing the *data access overhead.* At the same time, it degrades the performance of the hierarchical index by increasing the *index search overhead* (*i.e.*, the curse of dimensionality). As we know, the overall search performance depends largely on these two factors. By varying the size of the feature vector, we were able to identify the best case for each data set. Unless stated otherwise, the results shown in this section represent the best observed performance for the Haar and APCA transformations.

The VA-File, A-Tree, and *SCoBE* approximations do not reduce dimensionality explicitly. Instead, the size of the data representation is reduced by reducing the resolution of each value (*i.e.*, quantizing it). In our experiments, we used approximations of $\frac{1}{8}$ the size of the data objects.

We used two different resolutions of approximation with respect to the size of the data object. We used approximations of $\frac{1}{8}$ and $\frac{1}{16}$ the size of the data objects. Unless stated otherwise, performance comparisons were conducted using $\frac{1}{8}$ approximations.

6.4.4 Experimental Results on Whole Sequence Match Queries

In our experiments, we extensively evaluated the performance of the different k-NN search techniques described in Section 6.4.3 for whole sequence match. We measured the performance of these techniques using different data sets, different values of k, and different lengths of the time series. The number of objects in each data set remained fixed at 99,000 entries. We also compared *SCoBE* with techniques developed for evaluating dynamic time warping and subsequence match queries. The results of these experiments are presented toward the end of this section.

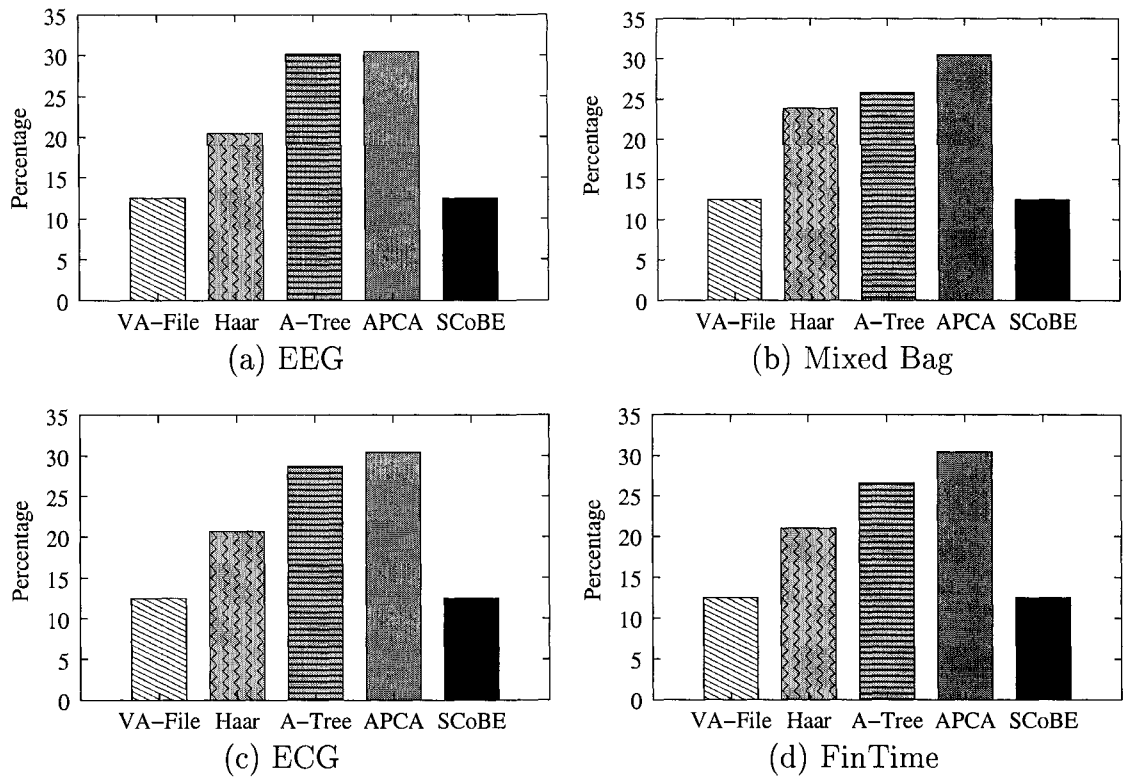


FIGURE 6.8. Index to Data Ratio. The Size of Hierarchical Indexes was Significantly larger than that of Linear Indexes

Index to Data Ratio: Figure 6.8 shows the *index to data ratio* for four different data sets due to the index used for processing k-NN search. To provide a fair comparison, each data approximation used the same amount of storage. Therefore, the difference in index sizes was due to the overhead caused by the internal organization of the hierarchical index. Since the VA-File and *SCoBE* use a linear index, they had the same size and this size was constant for all data sets. We observed larger storage requirements for the methods based on a hierarchical index (*i.e.*, Haar, A-Tree, and APCA). The APCA technique produced the largest storage overhead (about 30% the size of the data set) and this behavior was consistent for all data sets. In contrast, we observed that techniques based on a linear index (*i.e.*, VA-File, *SCoBE*) minimized the *index to data ratio.*

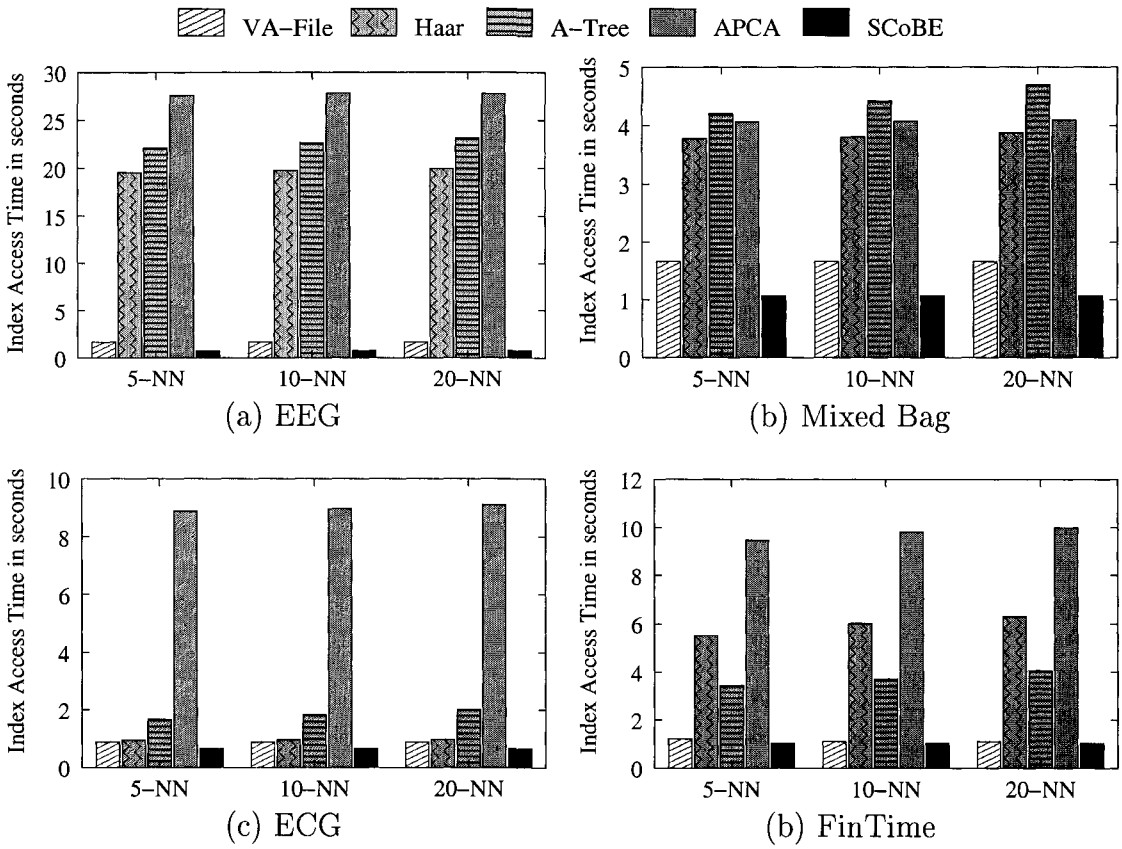


FIGURE 6.9. Index Search Overhead Shown as the Time Spent on Index IO. The Time Spent Searching a Hierarchical Index was Significant

Index Search Overhead: In Section 6.1, we indicated that some of the methods for similarity search are not always effective due to the large amount of time they might spend accessing the index. In Figure 6.9, we present our observations on the *Index Search Overhead* for all the techniques used in this study. The bars in Figure 6.9 show the time spent during index IO. To compute the index IO time, we first obtained the total IO time as the difference between elapsed time and CPU time. The index IO time is obtained from the total IO time by subtracting the time required to retrieve time series objects from the database.

In Figure 6.9, we observe that using a hierarchical index can have a negative effect in the search performance. In particular, the performance of the Haar, A-Tree, and APCA suffered for the EEG, Mixed Bag, and FinTime data sets due to the high *index*

search overhead. In contrast, the *index search overhead* for the techniques based on a linear index (*i.e.*, VA-File and *SCoBE*) was minimal and consistent for all data sets. We should remember that a hierarchical index is accessed using random IO. Therefore, if a large portion of the hierarchical index is accessed during similarity search, the performance benefits of having an index structure are nullified due to the high cost of random IO operations. On the other hand, a linear index (*e.g.*, like the index used by *SCoBE*) is accessed sequentially and this access pattern takes advantage of fast sequential IO operations.

Data Objects Fetched: The quality of an approximation can be measured by the number of data *objects fetched* during search. Tight bounding distances provided by the approximation will have a positive effect on reducing the *data search space.* In our experiments, we measured the number of data objects fetched for answering exact k-NN search queries. In Table 6.3, we present the number of objects that each technique fetched during 10-NN search. This table shows the results on all data sets with time series objects of length 256 and 1024. Note that the number of data pages accessed during search can be estimated from the number of objects fetched. One 8-KByte page can store one data time series of length 1024 or 4 data time series of length 256. Therefore, the number of data pages read equals the number of objects of length 1024 fetched. For objects of length 256, the following expression holds.

$$ObjectsFetched \geq DataPagesRead \geq ObjectsFetched/4$$

For queries requiring only the object ID, the number of objects fetched by *SCoBE* could be smaller than k (such is the case shown in Table 6.3). This is because the upper- and lower-bound distances can be used to verify if an object belongs to the k-NN set without fetching its data from disk (*i.e.*, step 2 of Algorithm 5). For queries that need to retrieve the data time series (*e.g.*, for further processing), this optimization cannot be used and the number of objects fetched is at least k. The

Data set	Objects Fetched (data length is 256)				
	VA-File	Haar	A-Tree	APCA	SCoBE
Plasma	1,488	1,748	106	9,355	7*
EEG	357	1,061	66	8,715	7*
ECG	16	283	27	131	14
Mixed Bag	16,371	2,197	36	3,200	17
FinTime	36	69	37	247	7*

Data set	Objects Fetched (data length is 1024)				
	VA-File	Haar	A-Tree	APCA	SCoBE
Plasma	2,062	13,798	123	41,530	6*
EEG	NA**	NA	NA	NA	NA
ECG	5*	1,508	16	997	5*
Mixed Bag	2,285	82	24	1,421	1*
FinTime	29	409	30	394	4*

* If only object ID is needed, k otherwise.

** Not Available, the EEG data set contains only objects of length 256.

TABLE 6.3. Number of Data Objects Fetched for a 10-NN Query

numbers shown here are for queries requiring only the IDs of the objects in the exact k-NN set.

From Table 6.3, we can observe that our proposed approach, *SCoBE*, fetched the smallest number of time series objects on all data sets. In contrast, the VA-File, Haar, and APCA techniques presented wide variations in the number of data *objects fetched* across different data sets and showed poor performance for the Plasma, EEG, and Mixed Bag data sets.

Elapsed Time: To this point, we have presented performance metrics aimed to understand the behavior of the different similarity search techniques used in this experimental study. In this section, we conclude our experimental evaluation of whole sequence match queries by presenting the performance as perceived by the user, and show the elapsed time measured by wall-clock. We loaded irrelevant data into the

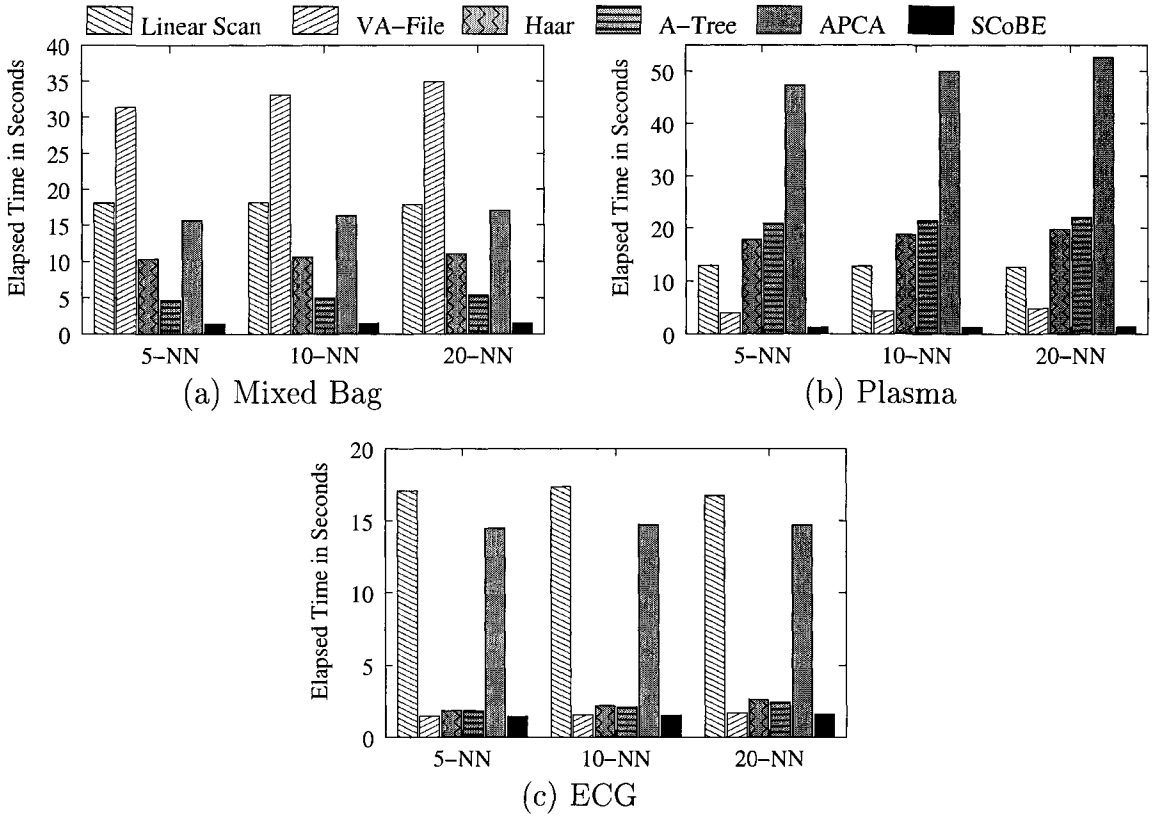


FIGURE 6.10. Elapsed Time of k-NN Search. Data Length on all Data Sets is 256

system to flush the entire memory between consecutive queries and avoid caching effects. Our results are summarized in Figures 6.10, 6.11, and Table 6.4.

Figure 6.10 shows the elapsed time on three data sets with time series objects of length 256. Each graph in this figure shows the elapsed times for k-NN search using all the techniques described in Section 6.4.3. In the experiments, we used k values of 5, 10, and 20. From Figure 6.10, we can observe that our proposed technique, *SCoBE*, outperformed the rest of the methods. *SCoBE* showed clear superiority for the Mixed Bag and Plasma data sets. While *SCoBE* provided the best performance for the ECG data set, the VA-File, Haar, and A-Tree approaches also showed good performance. The behavior observed in Figure 6.10 was expected. A poor *elapsed time* was observed on those techniques with either a significant *index search overhead* (Figure 6.9), or a large number of data *objects fetched* (Table 6.3), or the combination

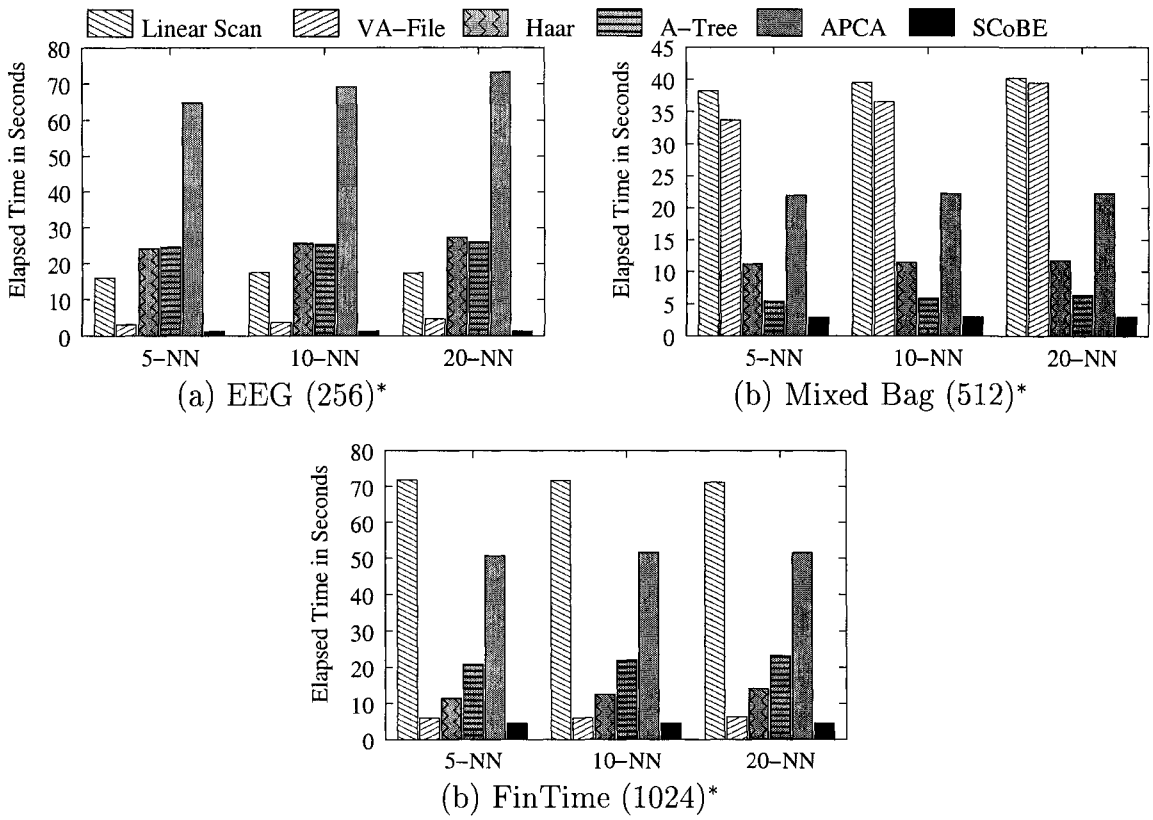


FIGURE 6.11. Elapsed Time of k-NN Search. *The Number Inside () Indicates Data Length

of both. The best elapsed time, on the other hand, was observed for *SCoBE*, which combines both a small *index search overhead* and a small number of data *objects fetched.*

In Figure 6.11, we show the elapsed time for the EEG, Mixed Bag, and FinTime data sets with time series objects of length 256, 512, and 1024, respectively. These results are similar to those shown in Figure 6.10. *SCoBE* showed the best elapsed time because it combines a minimal *index search overhead* and a small number of data *objects fetched.*

In a final experiment for whole sequence match queries, we reduced the size of the approximation and observed changes in the performance behavior. The size of the approximation used in this experiment was $\frac{1}{16}$ of the size of the data object. We

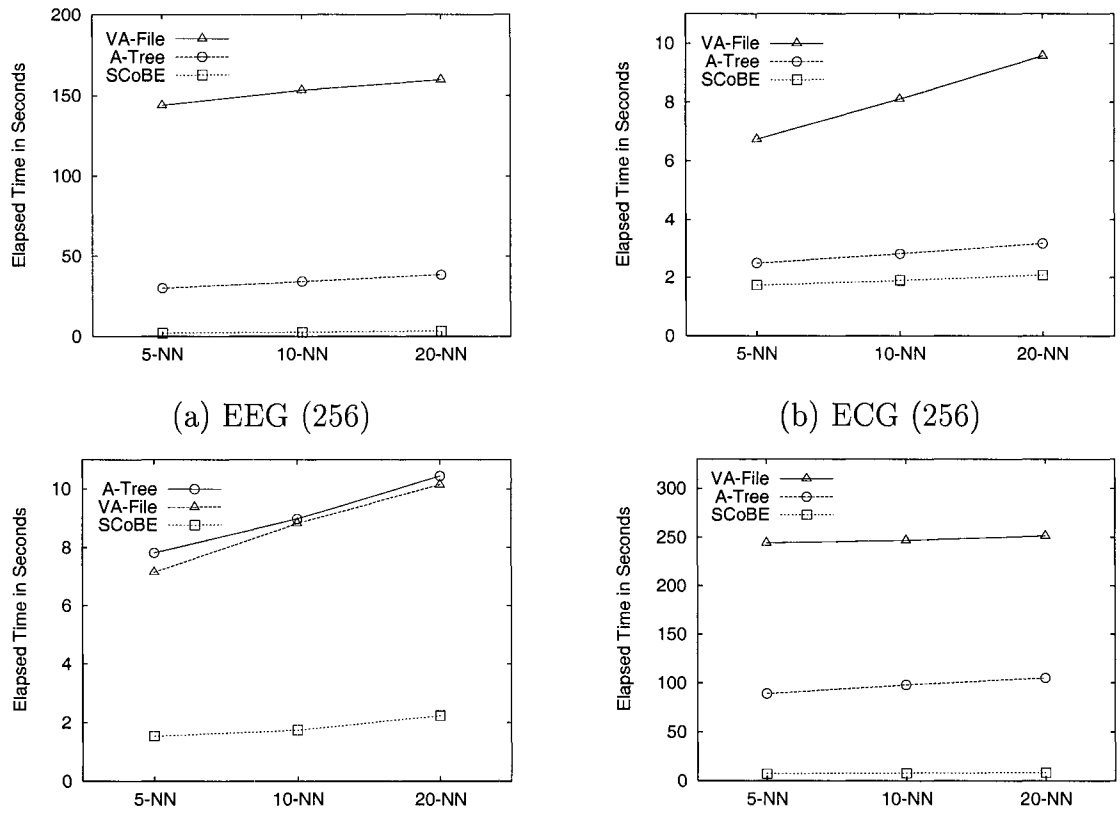


FIGURE 6.12. Elapsed Time of k-NN Search. Data Approximation is $\frac{1}{16}$ of the Size of the Data Object

excluded linear scan, Haar, and APCA from this experiment since they showed poor performance in the previous experiments. The effect of reducing the approximation size is twofold. First by reducing the size of the approximation, we reduce the index size. Second, a smaller-sized approximation may not be able to provide a tight lower-bound distance, thereby increasing the number of *data objects fetched.* Figure 6.12 shows the elapsed time observed for this experiment. Similar to the results shown in Figures 6.10 and 6.11, *SCoBE* significantly outperformed the A-Tree and VA-File.

Table 6.4 provides a clear picture of the performance of similarity search for all data sets with objects of length 256. In this table, we only included the VA-File

Data set (256)	Elapsed Time (seconds)		
	VA-File	A-Tree	SCoBE
Plasma	4.368	21.423	**1.320**
EEG	3.732	25.386	**1.320**
ECG	1.557	2.109	**1.497**
Mixed Bag	33.150	4.993	**1.467**
FinTime	1.759	4.202	**1.150**

TABLE 6.4. Elapsed Time for a 10-NN Query

and A-Tree because, in general, they performed better than the Haar and APCA techniques. Table 6.4 shows that our proposed technique, *SCoBE*, provided the best overall performance for all data sets in our experiments. Most importantly, the performance of the *SCoBE* was consistent across all data sets. In contrast, we observed wide variations in the performance for the VA-File and A-Tree across data sets.

6.4.5 Experimental Results on Dynamic Time Warping (DTW) Queries

We have evaluated DTW queries using *SCoBE* and compared our results to those obtained using Keogh's envelope. Because this idea was recently improved by Zhu and Shasha [137], we also include Zhu and Shasha's improved envelope in our empirical evaluation. In our experiments, we created an envelope around the query time series by limiting the warping path using the Sakoe-Chiba band [107]. When comparing two time series, this band restricts the displacement that a particular value in the series can have along the time dimension. Given two time series $\overline{q}$ and $\overline{s}$, this band indicates that value $\overline{q}[t]$ in $\overline{q}$ can only be compared to values $\overline{s}[t-r]$ to $\overline{s}[t+r]$ in $\overline{s}$, where $\overline{q}[t]$ is the value of time series $\overline{q}$ at time t, and r is time displacement allowed by the band.

In our experiments, we set the Sakoe-Chiba band to allow the time series to stretch about 10% on the time line (*i.e.*, $\pm 5\%$). We used all our data sets for this experiments but limited the length of the time series to 256. For Keogh's and Zhu's approaches, we

Data set	Objects Fetched		
	Envelope	**Improved E.**	**SCoBE**
Plasma	29,928	25,338	22,119
EEG	23,965	19,000	16,369
ECG	1,063	468	188
Mixed Bag	4,358	4,080	3,580
FinTime	275	209	119

Data set	Elapsed Time (seconds)		
	Envelope	**Improved E.**	**SCoBE**
Plasma	124.323	107.423	56.492
EEG	152.377	123.649	66.638
ECG	6.404	3.617	2.422
Mixed Bag	21.919	17.714	11.036
FinTime	6.126	3.341	2.147

TABLE 6.5. Observed Performance for 10-NN Dynamic Time Warping Queries

applied the Piecewise Aggregate Approximation (PAA) to every entry in the data set and inserted the corresponding data approximation into a disk-resident R-tree. For each data set, we created three different indexes using 8, 16, and 32 PAA segments, respectively. For Keogh's and Zhu's approaches, we evaluated k-NN queries using each of these indexes. However, we only show the results from the combination of data set and index with the best observed performance.

Our results for 10-NN queries are shown in Table 6.5. These are the averaged results of 100 queries. To prevent caching effects by the operating system, we loaded irrelevant data into the system to flush the entire memory between consecutive queries. Table 6.5 shows that *SCoBE* outperformed the state of the art techniques for DTW queries. Again, this was the result of combining efficient accesses to the index (being sequentially read) with an approximation that required fetching a smaller number of data objects than other methods.

6.4.6 Experimental Results on Subsequence Match Queries

We are aware of three index-based techniques for evaluating subsequence match queries, namely the *ST-index* [44], *DualMatch* [90], and *GeneralMatch* [89]. For brevity, and because GeneralMatch has been shown to outperform the other techniques [89], we compare *SCoBE* only with GeneralMatch and linear scan.

We implemented GeneralMatch in the same way described by its authors. We extracted all subsequences from the data time series using a step of j time units. Because the value of j affects the performance GeneralMatch for evaluating subsequence match queries, it is necessary to experiment with several values of j. After trying several values of j, for each data set we selected the value of j that resulted in the best observed performance. For each subsequence, we applied DFT and used six coefficients to create a feature vector that we inserted into an R-tree. In our performance evaluation, we used the Plasma, ECG, and Mixed Bag data sets as a single, long, time series. The resulting lengths were 558,080, 430,080, and 1,393,664, respectively. For the FinTime data set, we used 99,000 data time series of length 1024 each. We excluded the EEG data set because it only contains short sequences (*i.e.*, of length 256).

Table 6.6 shows averaged results from executing 100 10-NN queries. We used query subsequences of length 256. These results are consistent with our previous observations that IO overhead severely affects search performance. *SCoBE* outperformed GeneralMatch and linear scan because it accessed the minimum number of disk pages. It sequentially accessed a linear index which is only a fraction of the size of the data and it only fetched a small number of subsequences to guarantee the correctness of the query result. For the FinTime data set, *SCoBE* accessed a larger number pages than GeneralMatch. However, 24,751 of these pages were from the linear index, which was sequentially read.

Data set	Subsequences Fetched			Pages Accessed		
	GMatch	L. Scan	SCoBE	GMatch	L. Scan	SCoBE
Plasma	148,816	557,825	30	475	545	167
ECG	32,805	429,825	34	323	420	140
Mixed Bag	87,944	1,393,409	15	383	1,361	356
FinTime	134,913	76,131,000	20	11,452	99,000	24,771

Data set	Elapsed Time (seconds)		
	GeneralMatch	Linear Scan	SCoBE
Plasma	3.560	5.210	0.931
ECG	1.61	4.007	0.493
Mixed Bag	2.562	12.986	1.146
FinTime	114.744	719.728	41.78

TABLE 6.6. Observed Performance for 10-NN Subsequence Match Queries

6.5 Summary of Results

In 1994, Faloutsos *et al.* [44] suggested that any newly proposed technique for evaluating similarity search queries should satisfy a set of desirable properties. Namely, the new technique should be faster than linear scan, it should use little storage overhead, it should handle queries of arbitrary length, it should gracefully handle insertions and deletions, and it should have no false dismissals.

In this chapter, we have presented the *Self COntained Bit Encoding (SCoBE)*, a simple technique for representing time series data in a compact format. *SCoBE* satisfies all of the properties enumerated by Faloutsos. In addition, we have experimentally shown that *SCoBE*, organized in a linear index, outperforms sophisticated transformations and hierarchical indexes during exact k-NN search for both whole sequence and subsequence match queries. We have provided experimental evidence showing that, contrary to state-of-the-art techniques for similarity search on time series data, our proposed technique provided a consistently good performance under a variety of settings. This consistency was observed on queries with and without time warping and for the four performance metrics used: *index storage overhead*, *index*

search overhead, *data objects fetched*, and *elapsed time*.

The effectiveness of *SCoBE* for k-NN search on time series data comes from minimizing the *index search overhead* as well as from reducing the number of *data objects fetched*. By organizing the *SCoBE* approximations in a linear index, we minimize the storage for the index. Most importantly, we can take advantage of fast sequential disk accesses. In consequence, we are able to reduce the time spent during index search. In addition, our *SCoBE* approximation provides tight upper- and lower-bound distances to the data objects. This allows us to filter out a large number of irrelevant entries and to drastically reduce the data search space.

Chapter 7
Conclusions and Future Work

In data analysis applications we look for interesting patterns in data. For this, we formulate queries that summarize the information contained in large databases [53]. In traditional databases, this is a challenging problem because we are often dealing with vast amounts of data. In such case, aggregation is used to reduce the amount of information that a human needs to analyze before making a decision. When we consider time-varying information, the complexity of the queries that summarize information increases because temporal databases are larger in size than their non-temporal counterparts and also because the temporal ordering and validity of every entry in the database must be considered.

In this dissertation we have studied and proposed effective and efficient solutions to the problem of summarizing time-evolving data from two different perspectives. First, we studied the problem of capturing the collective behavior of time-evolving data. For this problem, known as temporal aggregation, we propose a model to better understand temporal aggregation queries. In addition, we introduce new IO efficient algorithms for the evaluation of these queries. We have also studied the problem of identifying entries in the database following a particular temporal pattern. For this problem, known as similarity search, we have proposed techniques for indexing and representing time series data. These techniques can be used for efficiently evaluating similarity search queries.

The proposed algorithms for computing temporal aggregation provide significant benefits over the current state of the art in different ways. We have developed new sequential and parallel bucket algorithms based on novel data partitioning schemes. These algorithms can be used to compute temporal aggregates for databases that are

substantially larger than the size of available memory, by processing data partitions in a sequential or parallel fashion. In particular, with the adaptive data partitioning scheme and the local and global meta arrays for partitioned data, we have demonstrated that the parallel bucket algorithm achieves scalable performance for large-scale databases by delivering nearly linear scale-up and speed-up, even at the presence of data skew.

We have observed that there are a few factors that affect the performance of temporal aggregation queries. These factors include the percentage of long-lived tuples and the number of buckets used for data partitioning. Although the proposed algorithms outperformed previous approaches consistently irrespective of such conditions, we believe it is worth elaborating further on the issues. Additionally, we plan to study performance impacts of such factors as initial data placement (*e.g.*, temporal partitioning vs. non-temporal partitioning) and data reduction by aggregation. We also plan to extend the data partitioning approach to spatio-temporal databases, which requires computing aggregates for data objects with two or more dimensional extents. Unlike the temporal aggregation, we expect that the process of data partitioning and generating meta arrays will be more sophisticated.

One of our contributions for the efficient evaluation of similarity search queries is the *Skyline Index*, a simple and elegant paradigm for indexing time series data. The Skyline index uses Skyline Bounding Regions (SBRs) to organize time series data in a hierarchical data structure. We have shown that the skyline index can be easily implemented using popular indexing libraries such as GiST [60], which demonstrates the practical feasibility of our idea. In the experimental results, we show that the Skyline Index can be coupled with state of the art dimensionality reduction techniques such as APCA [74] and significantly improve the performance on the evaluation of similarity search queries.

For future work, we plan to study further other approximation techniques, and the effects of different splitting algorithms for the Skyline Index construction. We

also plan to apply the Skyline Index approach to different dimensionality reduction techniques (*e.g.*, DWT and DFT) to further examine the applicability of this new indexing paradigm.

A second contribution for the evaluation of similarity search queries is the *Self COntained Bit Encoding (SCoBE)*, a simple technique for representing time series data in a compact format. We have experimentally shown that *SCoBE* outperforms sophisticated transformations for the evaluation of similarity search queries. We have provided experimental evidence showing that *SCoBE* has a consistently good performance under a variety of settings. This consistency was observed on queries with and without time warping. The effectiveness of *SCoBE* for k-NN search on time series data comes from minimizing the index search overhead as well as from reducing the number of data objects fetched. By organizing the *SCoBE* approximations in a linear index, we minimize the storage for the index. Most importantly, we can take advantage of fast sequential disk accesses. In consequence, we are able to reduce the time spent during index search. In addition, our *SCoBE* approximation provides tight upper- and lower-bound distances to the data objects. This allows us to filter out a large number of irrelevant entries and to drastically reduce the data search space.

The experiments shown in this dissertation were performed for uni-variate time series. That is, only one attribute of interest was considered during similarity search. In the future, we plan to explore the performance behavior of *SCoBE* when used for the evaluation of similarity search queries on multi-variate time series. Our experiments were also limited to the use of Euclidean distance as our similarity metric. We did this because some of the methods being compared could only support this metric. It is our interest to evaluate the performance of *SCoBE* when other similarity models are used.

References

[1] Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, August 2003.

[2] Sameet Agarwal, Rakesh Agrawal, Prasad M. Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the Computation of Multidimensional Aggregates. In *Proceedings of the VLDB Conference*, pages 506–521, Bombay, India, September 1996.

[3] Charu C. Aggarwal. On the Effects of Dimensionality Reduction on High Dimensional Similarity Search. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 256–266, Santa Barbara, CA, May 2001.

[4] Rakesh Agrawal, Christos Faloutsos, and Arun N. Swami. Efficient Similarity Search in Sequence Databases. In *Proceedings of the FODO Conference*, pages 69–84, Evansotn, IL, October 1993.

[5] Rakesh Agrawal, King-Ip Lin, Harpreet S. Sawhney, and Kyuseok Shim. Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases. In *Proceedings of the VLDB Conference*, pages 490–501, Zurich, Switzerland, September 1995.

[6] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, Madison, WI, June 2002. Invited talk.

[7] Mike Barnett, Satya Gupta, David G. Payne, Lance Shuler Robert van de Geijn, and Jerrel Watts. Interprocessor Collective Communication Library (InterCom). In *Proceedings of the IEEE Scalable High Performance Computing Conference*, pages 357–364, Knoxville, TN, May 1994.

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R^*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the ACM-SIGMOD Conference*, pages 322–331, Atlantic City, NJ, May 1990.

[9] Richard Bellman. *Adaptive Control Processes: A Guided Tour.* Princeton University Press, 1961.

[10] Jon L. Bentley. Algorithms for Klee's Rectangle Problems. Technical Report unpublished, Pittsburgh, PA, 1977.

[11] Stefan Berchtold, Christian Bohm, H. V. Jagadish, Hans-Peter Kriegel, and Jorg Sander. Independent Quantization: An Index Compression Technique for High-Dimensional Data Spaces. In *Proceedings of the International Conference on Data Engineering*, pages 577–588, San Diego, CA, 28 February – 3 March 2000. IEEE Computer Society.

[12] Stefan Berchtold, Christian Bohm, Daniel Keim, Florian Krebs, and Hans-Peter Kriegel. On Optimizing Nearest Neighbor Queries in High-Dimensional Data Spaces. In *Proceedings of the International Conference in Database Theory*, pages 435–449, London, UK, January 2001.

[13] Donald J. Berndt and James Clifford. Using Dynamic Time Warping to Find Patterns in Time Series. In *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases*, pages 359–370, Seattle, WA, July 1994.

[14] Elisa Bertino, Beng U. Oo abd Ron Sacks-Davis, Kin-Lee Tan, Justin Zobel, Boris S Shidlovsky, and Barbara Catania. *Indexing Techniques for Advanced Database Systems.* Kluwer Academic Publishers, Boston, MA, 1997.

[15] Claudio Bettini, Curtis E. Dyreson, William S. Evans, Richard T. Snodgrass, and Xiaoyang Sean Wang. *Temporal Databases: Research and Practice*, chapter A Glossary of Time Granularity Concepts, pages 406–413. Springer, 1998.

[16] Claudio Bettini, Sushil Jajodia, and Sean X. Wang. *Time Granularities in Databases, Data Mining, and Temporal Reasoning.* Springer, Berlin, 2000.

[17] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "Nearest Neighbor" Meaningful? In *Proceedings of the International Conference in Database Theory*, pages 217–235, Jerusalem, Israel, January 1999.

[18] Luca Cabibbo and Riccardo Torlone. A Framework for the Investigation of Aggregate Functions in Database Queries. In *Proceedings of the International Conference in Database Theory*, pages 383–397, Jerusalem, Israel, January 1999.

[19] Ohio Supercomputer Center. LAM/MPI Parallel Computing. `http://www.osc.edu/lam.html`, 1998.

[20] Kaushik Chakrabarti and Sharad Mehrotra. The Hybrid Tree: An Index Structure for High Dimensional Feature Spaces. In *Proceedings of the International Conference on Data Engineering*, pages 440–447, Sydney, Australia, March 1999. IEEE Computer Society.

[21] Kin-Pong Chan and Ada Wai-Chee Fu. Efficient Time Series Matching by Wavelets. In *Proceedings of the International Conference on Data Engineering*, pages 126–133, Sydney, Australia, March 1999. IEEE Computer Society.

[22] King-Pong Chan, Ada Wai-Chee Foo, and Clement Yu. Haar Wavelets for Efficient Similarity Search of Time-Series: With and Without Time Warping. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):686–705, May–June 2003.

[23] Surajit Chaudhuri, Gautam Das, Mayur Datar, Rajeev Motwani, and Vivek Narasayya. Overcoming Limitations of Sampling for Aggregation Queries. In *Proceedings of the International Conference on Data Engineering*, pages 534–542, Heidelberg, Germany, April 2001. IEEE Computer Society.

[24] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. A Robust, Optimization-Based Approach for Approximate Answering of Aggregate Queries. In *Proceedings of the ACM-SIGMOD Conference*, pages 295–306, Santa Barbara, CA, May 2001.

[25] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, March 1997.

[26] Surajit Chaudhuri and Kyuseok Shim. Including Group By in Query Optimization. In *Proceedings of the VLDB Conference*, pages 354–366, Santiago, Chile, September 1994.

[27] Edgar Chavez, Gonzalo Navarro, Ricardo A. Baeza-Yates, and Jose L. Marroquin. Searching in Metric Spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[28] Tsz S. Cheng and Shashi K. Gadia. A Pattern-matching Language for Spatio-Temporal Databases. In *Proceedings of the ACM-CIKM Conference*, pages 280–287, Gaithersburg, MD, November 29 – December 2 1994.

[29] Selina Chu, Eamonn Keogh, David Hart, and Michael Pazzani. Iterative Deepening Dynamic Time Warping for Time Series. In *Proceedings of the 2nd SIAM International Conference on Data Mining*, Arlington, VA, April 2002.

[30] Seok-Ju Chun, Chin-Wan Chung, Ju-Hong Lee, and Seok-Lyong Lee. Dynamic Update Cube for Range-Sum Queries. In *Proceedings of the VLDB Conference*, pages 521–530, Roma, Italy, September 2001.

[31] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the VLDB Conference*, pages 206–215, Athens, Greece, September 1997.

[32] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Equivalences Among Aggregate Queries with Negation. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 215–226, Santa Barbara, CA, May 2001.

[33] COHOWeb. Deep space hourly merged magnetic field, plasma, and ephemerides data. `http://nssdc.gsfc.nasa.gov/cohoweb/cw.html`, October 2002.

[34] Jeffrey Considine, Feifei Li, George Kollios, and John Byers. Approximate Aggregation Techniques for Sensor Databases. In *Proceedings of the International Conference on Data Engineering*, pages 449–460, Boston, MA, March 30–April 2 2004. IEEE Computer Society.

[35] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* McGraw Hill, Cambridge, MA, 1990.

[36] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining Streams Statistics over Sliding Windows (Extended Abstract). In *Proceedings of the annual ACM-SIAM Symposium on Discrete Algorithms*, pages 635–644, San Francisco, CA, January 2002.

[37] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the ACM-SIGMOD Conference*, pages 1–8, Boston, MA, June 1984.

[38] Alin Dobra, Minos Garofalakis, Johanes Gehrke, and Rajeev Rastogi. Processing Complex Aggregate Queries over Data Streams. In *Proceedings of the ACM-SIGMOD Conference*, pages 61–72, Madison, WI, June 2002.

[39] Marlon Dumas, Marie-Christine Fauvet, and Pierre-Claude Scholl. Handling Temporal Grouping and Pattern-matching Queries in a Temporal Object Model. In *Proceedings of the ACM-CIKM Conference*, pages 424–431, Bethesda, MD, November 1998.

[40] Curtis E. Dyreson, William S. Evans, Hong Lin, and Richard T. Snodgrass. Efficiently Supporting Temporal Granularities. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):565–587, July–August 2000.

[41] Robert Epstein. Techniques for Processing of Aggregates in Relational Database Systems. Technical Report UCB/ERL M7918, University of California, Berkeley, CA, February 1979.

[42] Christos Faloutsos. *Searching Multimedia Databases By Content.* Kluwer Academic Publishers, Boston, MA, 1996.

[43] Christos Faloutsos and King-Ip Lin. FastMap: A Fast Algorithm for Indexing Data-Mining and Visualization of Traditional and Multimedia Datasets. In *Proceedings of the ACM-SIGMOD Conference*, pages 163–174, San Jose, CA, May 1995.

[44] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast Subsequence Matching in Time-Series Databases. In *Proceedings of the ACM-SIGMOD Conference*, pages 419–429, Minneapolos, MN, May 1994.

[45] Hakan Ferhatosmanoglu, Ertem Tuncel, D. Agrawal, and A. E. Abbadi. Vector Approximation based Indexing for Non-uniform Dimensional Data Sets. In *Proceedings of the ACM-CIKM Conference*, pages 202–209, McLean, VA, November 2000.

[46] Johann C. Freytag and Nathan Goodman. Translating Aggregate Queries into Iterative Programs. In *Proceedings of the VLDB Conference*, pages 138–146, Kyoto, Japan, August 1986.

[47] Steven Geffner, Divakant Agrawal, and Amr El Abbadi. The Dynamic Data Cube. In *Proceedings of the Conference on Extending Database Technology*, pages 237–253, Konstanzi, Germany, March 2000.

[48] Jose Alvin G. Gendrano, Bruce C. Huang, Jim M. Rodrigue, Bongki Moon, and Richard T. Snodgrass. Parallel Algorithms for Computing Temporal Aggregates. In *Proceedings of the International Conference on Data Engineering*, pages 418–427, Sydney, Australia, March 1999. IEEE Computer Society.

[49] Anna C. Gilbert, Yannis Lotidis, S Muthukrishnan, and Martin J. Strauss. Optimal and Approximate Computation of Summary Statistics for Range Aggregates. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 227–236, Santa Barbara, CA, May 2001.

[50] Anna C. Gilbert, Yannis Lotidis, S. Muthukrishnan, and Martin J. Strauss. Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. In *Proceedings of the VLDB Conference*, pages 79–88, Roma, Italy, September 2001.

[51] Dina Q. Goldin and Paris C. Kanellakis. On Similarity Queries for Time-Series Data: Constraint Specification and Implementation. In *Proceedings of the 1st International Conference on the Principles and Practice of Constraint Programming*, pages 36–45, Cassis, France, September 1995.

[52] Jim Gray. *The Benchmark Handbook for Database and Transaction Processing Systems.* Morgan Kaufmann, 1991.

[53] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[54] Stephane Grumbach and Leonardo Tininini. On the Content of Materialized Aggregate Views. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 47–57, Dallas, TX, May 2000.

[55] Sudipto Guha, Nick Koudas, and Kyuseok Shim. Data-Streams and Histograms. In *Proceedings of the annual ACM Symposium on Theory of Computing*, pages 471–475, Hersonissos, Crete, Greece, July 2001.

[56] Antonin Guttman. R-Trees: a Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM-SIGMOD Conference*, pages 47–57, Boston, MA, June 1984.

[57] Peter J. Haas and Joseph M. Hellerstein. Ripple Joins for Online Aggregation. In *Proceedings of the ACM-SIGMOD Conference*, pages 287–298, Philadelphia, PA, June 1999.

[58] James P. Held and John V. Carlis. Match: A new High-level Relational Operator for Pattern Matching. *Communications of the ACM*, 30(1):62–75, January 1987.

[59] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online Aggregation. In *Proceedings of the ACM-SIGMOD Conference*, pages 171–182, Tucson, AZ, May 1997.

[60] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings of the VLDB Conference*, pages 562–573, Zurich, Switzerland, September 1995.

[61] S. Hettich and S. D. Bay. The UCI KDD Archive. `http://kdd.ics.uci.edu`, 2002.

[62] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. Range Queries in OLAP Data Cubes. In *Proceedings of the ACM-SIGMOD Conference*, pages 73–88, Tucson, AZ, May 1997.

[63] Kaippallimalil J. Jacob and Dennis Shasha. FinTime – a Financial Time Series Benchmark. `http://cs.nyu.edu/cs/faculty/shasha/fintime.html`, March 2000.

[64] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth A. Sevcik, and Torsten Suel. Optimal Histograms with Quality Guarantees. In *Proceedings of the VLDB Conference*, pages 275–286, New York, USA, August 1998.

[65] Joseph JaJa, Steve Kelley, and Dave Rafkind. Information Discovery in Spatio-Temporal Environmental Data. Technical Report unpublished, 2002.

[66] Christian S. Jensen and Richard T. Snodgrass. Semantics of Time-Varying Information. *Information Systems*, 21(4):311–352, June 1996.

[67] Richard A. Johnson and Dean W. Wichern. *Applied Multivariate Analysis.* Prentice Hall, Englewood Cliffs, NJ, 1982.

[68] Tamer Kahveci and Ambij K. Singh. Variable Length Queries for Time Series Data. In *Proceedings of the International Conference on Data Engineering*, pages 273–282, Heidelberg, Germany, April 2001. IEEE Computer Society.

[69] Eamon Keogh. Exact Indexing of Dynamic Time Warping. In *Proceedings of the VLDB Conference*, pages 406–417, Hong Kong, China, August 2002.

[70] Eamon Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. *Knowledge and Information Systems*, 3(3):263–286, 2000.

[71] Eamon Keogh and Shruti Kasetty. On the Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration. In *Proceedings of the ACM-SIGKDD Conference*, pages 102–111, Edmonton, Alberta, Canada, July 2002.

[72] Eamon Keogh and Michael Pazzani. Scaling up Dynamic Time Warping for Datamining Applications. In *Proceedings of the ACM-SIGKDD Conference*, pages 285–289, Boston, MA, August 2000.

[73] Eamon Keogh and Padhraic Smyth. A Probabilistic Approach to Fast Pattern Matching in Time Series Databases. In *Proceedings of the 3rd International Conference of Knowledge Discovery and Data Mining*, pages 24–30, Newport Beach, CA, August 1997.

[74] Eamonn Keogh, Kaushik Chakrabarti, Sharad Mehrotra, and Michael Pazzani. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. In *Proceedings of the ACM-SIGMOD Conference*, pages 151–162, Santa Barbara, CA, May 2001.

[75] Eamonn Keogh and Michael Pazzani. An Indexing Scheme for Fast Similarity Search in Large Time Series Databases. In *Proceedings of the International Conference on Scientific and Statistical Database Management*, pages 56–67, Cleveland, OH, 1999.

[76] Jong S. Kim, Sung T. Kang, and Myoung-H. Kim. On Temporal Aggregate Processing based on Time Points. *Information Processing Letters*, 71(5-6):213–220, September 1999.

[77] Sang-Wook Kim, Sanghyun Park, and Wesley W. Chu. An Index-Based Approach for Similarity Search Supporting Time Warping in Large Sequence Databases. In *Proceedings of the International Conference on Data Engineering*, pages 607–614, Heidelberg, Germany, April 2001. IEEE Computer Society.

[78] Rodger N. Kline. *Aggregation in Temporal Databases.* PhD thesis, University of Arizona, Tucson, Arizona, May 1999.

[79] Rodger N. Kline and Richard T. Snodgrass. Computing Temporal Aggregates. In *Proceedings of the International Conference on Data Engineering*, pages 222–231, Taipei, Taiwan, March 1995. IEEE Computer Society.

[80] Athony Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *Journal of the ACM*, 29(3):699–717, July 1982.

[81] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley Publishing Company, Reading, MA, 1973.

[82] Flip Korn, H. V. Jagadish, and Christos Faloutsos. Efficiently Supporting Ad Hoc Queries in Large Datasets of Time Sequences. In *Proceedings of the ACM-SIGMOD Conference*, pages 289–300, Tucson, AZ, May 1997.

[83] Per-Ake Larson. Data Reduction by Partial Preaggregation. In *Proceedings of the International Conference on Data Engineering*, pages 706–715, San Jose, CA, February 26–March 1 2002. IEEE Computer Society.

[84] Udi Manber. *Introduction to Algorithms: A Creative Approach.* Addison-Wesley Publishing Company, Reading, MA, 1989.

[85] Yannis Manolopoulos, Yannis Theodoridis, and Vassilis J. Tsotras. *Advanced Database Indexing.* Kluwer Academic Publishers, Boston, MA, 2000.

[86] Jim Melton. *Advanced SQL:1999. Understanding Object-Relational and Other Advanced Features.* The Morgan Kaufman Series in Data Management Systems. Morgan Kaufmann Publishers, San Francisco, CA, 2003.

[87] George B. Moody. MIT-BIH Database Distribution. `http://ecg.mit.edu/index.html`, 1999.

[88] Bongki Moon, Ines F. Vega Lopez, and Vijaykumar Immanuel. Efficient Algorithms for Large-Scale Temporal Aggregation. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):744–751, May–June 2003.

[89] Yang-Sae Moon, Kyu-Young Whang, and Wook-Shin Han. General Match: A Subsequence Matching Method in Time-Series Databases Based on Generalized Windows. In *Proceedings of the ACM-SIGMOD Conference*, pages 382–393, Madison, WI, June 2002.

[90] Yang-Sae Moon, Kyu-Young Whang, and Woong-Kee Loh. Duality-Based Subsequence Matching in Time-Series Databases. In *Proceedings of the International Conference on Data Engineering*, pages 263–272, Heidelberg, Germany, April 2001. IEEE Computer Society.

[91] Yuu Morinaka, Masatoshi Yoshikawa, Toshiyuki Amagasa, and Shunsuke Uemura. The L-index: An Indexing Structure for Efficient Subsequence Matching in Time Sequence Databases. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 51–60, Kowloon, Hong Kong, April 2001.

[92] Peng Ning, Xiaoyang Sean Wang, and Sushil Jajodia. An Algebraic Representation of Calendars. *Annals of Mathematics and Artificial Intelligence*, 36(1-2):5–38, 2002.

[93] Sanghyun Park, Wesley W. Chu, Jeehee Yoon, and Chihcheng Hsu. Efficient Searches for Similar Subsequences of Different Lengths in Sequence Databases. In *Proceedings of the International Conference on Data Engineering*, pages 23–32, San Diego, California, 28 February - 3 March 2000. IEEE Computer Society.

[94] Chang-Shing Perng, Haixun Wang, Sylvia R. Zhang, and Douglas S. Parker. Landmarks: A New Model for Similarity-based Pattern Querying in Time Series Databases. In *Proceedings of the International Conference on Data Engineering*, pages 33–42, San Diego, CA, 28 February - 3 March 2000. IEEE Computer Society.

[95] Gregory Piatetsky-Shapiro and Charles Connel. Accurate Estimation of the Number of Tuples Satisfying a Condition. In *Proceedings of the ACM-SIGMOD Conference*, pages 256–276, Boston, MA, June 1984.

[96] Ivan Popivanov. Efficient Similarity Queries over Time Series Data using Wavelets. Master's thesis, University of Toronto, Toronto, Canada, 2001.

[97] Ivan Popivanov and Renee J. Miller. Similarity Search Over Time-Series Data Using Wavelets. In *Proceedings of the International Conference on Data Engineering*, pages 212–221, San Jose, CA, 26 February - 1 March 2002. IEEE Computer Society.

[98] Lin Qiao, Divy Agrawal, and Amr El Abbadi. RHist: Adaptive Summarization over Continuous Data Streams. In *Proceedings of the ACM-CIKM Conference*, pages 469–476, McLean VA, November 2002.

[99] Lawrence Rabiner and Biing-Hwang Juang. *Fundamentals of Speech Recognition.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[100] Davood Rafiei. On Similarity-Based Queries for Time Series Data. In *Proceedings of the International Conference on Data Engineering*, pages 410–417, Sydney, Australia, March 1999. IEEE Computer Society.

[101] Davood Rafiei and Alberto Mendelzon. Similarity-Based Queries for Time Series Data. In *Proceedings of the ACM-SIGMOD Conference*, pages 13–25, Tucson, AZ, May 1997.

[102] Davood Rafiei and Alberto Mendelzon. Efficient Retrieval of Similar Time Sequences Using DFT. In *Proceedings of the FODO Conference*, Kobe, Japan, November 1998.

[103] Sudha Ram. Intelligent Database Design Using the Unifying Semantinc Model. *Information and Management*, 29(1995):191–206, 1995.

[104] Sudha Ram and Veda C. Storey. Composite and Grouping: Extending the Realm of Semantic Modeling. In *Proceedings of the Hawaii International Conference on System Sciences HICSS*, pages 212–218, Maui, HA, January 1993.

[105] Raghu Ramakrishnan. *Database Management Systems.* WCB McGraw-Hill, 1998.

[106] Mirek Riedewald, Divyakant Agrawal, and Amr El Abbadi. Efficient Integration and Aggregation of Historical Information. In *Proceedings of the ACM-SIGMOD Conference*, pages 13–24, Madison, WI, June 2002.

[107] Hiroaki Sakoe and Seibi Chiba. Dynamic Programming Algorithm Optimization for Spoken Word Recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(1):43–49, February 1978.

[108] Yasushi Sakurai, Masatoshi Yoshikawa, Shunsuke Uemura, and Haruhiko Kojima. The A-Tree: An Index Structure for High-dimensional Spaces Using Relative Approximation. In *Proceedings of the VLDB Conference*, pages 516–526, Cairo, Egypt, September 2000.

[109] Betty Salzberg and Vassilis Tsotras. Comparison of Access Methods for Time-Evolving Data. *ACM Computing Surveys*, 31(2):158–221, June 1999.

[110] Thomas Seidl and Hans-Peter Kriegel. Optimal Multi-Step k-Nearest Neighbor Search. In *Proceedings of the ACM-SIGMOD Conference*, pages 154–165, Seattle, WA, May 1998.

[111] Manigantan Sethuraman. Implementation and Evaluation of a Partitioned Store for Transaction-Time Databases. Master's thesis, University of Arizona, Tucson, AZ, December 2003.

[112] Hagit Shatkay and Stanley B. Zdonik. Approximate Queries Representations for Large Data Sequences. In *Proceedings of the International Conference on Data Engineering*, pages 536–545, New Orleans, LA, February 1996. IEEE Computer Society.

[113] Abraham Silberschatz, Henry F. Kort, and S. Sudarshan. *Database System Concepts*. WCB McGraw-Hill, third edition, 1999.

[114] Richard T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, San Francisco, CA, 2000.

[115] Richard T. Snodgrass and Ilsoo Ahn. A Taxonomy of Time in Databases. In *Proceedings of the ACM-SIGMOD Conference*, pages 236–246, Austin, TX, May 1985.

[116] Richard T. Snodgrass and Ilsoo Ahn. Temporal Databases. *IEEE Computer*, 19(9):35–41, September 1986.

[117] Richard T. Snodgrass, Santiago Gomez, and L. Edwin McKenzie Jr. Aggregates in the Temporal Query Language TQuel. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):826–842, September–October 1993.

[118] Michael D. Soo, Richard T. Snodgrass, and Christian S. Jensen. Efficient Evaluation of the Valid-Time Natural Join. In *Proceedings of the International Conference on Data Engineering*, pages 282–292, Houston, TX, February 1994.

[119] Michael Stonebraker. The Case for Shared Nothing. *A Quarterly bulletin of the IEEE Computer Society Technical Committee on Database Engineering*, 9(1):4–9, March 1986.

[120] Zbigniew R. Struzik and Arno Siebes. The Haar Wavelet Transform in the Time Series Similarity Paradigm. In *Proceedings of the 3rd European Conference on Principles of Data Mining and Knowledge Discovery*, pages 12–22, Berlin, Germany, September 1999.

[121] Abdullah U. Tansel, James Clifford, and Shashi Et Al Gadia. *Temporal Databases: Theory, Design, and Implementation.* Database Systems and Applications. Benjamin Cummins, 1993.

[122] Yufei Tao, Dimitris Papadias, and Christos Faloutsos. Approximate Temporal Aggregation. In *Proceedings of the International Conference on Data Engineering*, pages 190–201, Boston, MA, March 30–April 2 2004. IEEE Computer Society.

[123] Paul A. Tuma. Implementing Historical Aggregates in TempIS. Master's thesis, Wayne State University, Detroit, Michigan, November 1992.

[124] Ertem Tuncel, Hakan Ferhatosmanoglu, , and Kenneth Rose. VQ-Index: An Index Structure for Similarity Searching in Multimedia Databases. In *Proceedings of the 10th ACM International Conference on Multimedia*, pages 543–552, Juan Les Pins, France, December 2002.

[125] Roger Weber and Klemens Bohm. Trading Quality for Time with Nearest-Neighbor Search. In *Proceedings of the Conference on Extending Database Technology*, pages 21–35, Konstanzi, Germany, March 2000.

[126] Roger Weber, Hans-Jorg Schek, and Stephen Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *Proceedings of the VLDB Conference*, pages 194–205, New York, USA, August 1998.

[127] Yi-Leh Wu, Divyakant Agrawal, and Amr El Abbadi. A Comparison of DFT and DWT Based Similarity Search in Time-Series Databases. In *Proceedings of the ACM-CIKM Conference*, pages 488–495, McLean, VA, November 2000.

[128] Weipeng P. Yan and Per-Ake Larson. Eager Aggregation and Lazy Aggregation. In *Proceedings of the VLDB Conference*, pages 345–357, Zurich, Switzerland, September 1995.

[129] Jun Yang and Jennifer Widom. Incremental Computation and Maintenance of Temporal Aggregates. In *Proceedings of the International Conference on Data Engineering*, pages 51–60, Heidelberg, Germany, April 2001. IEEE Computer Society.

[130] Xinfeng Ye and John A. Keane. Processing Temporal Aggregates in Parallel. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 1373–1378, Orlando, FL, October 1997.

[131] Byoung-Kee Yi and Christos Faloutsos. Fast Time Sequence Indexing for Arbitrary L_p Norms. In *Proceedings of the VLDB Conference*, pages 385–394, Cairo, Egypt, September 2000.

[132] Byoung-Kee Yi, H. V. Jagadish, and Christos Faloutsos. Efficient Retrieval of Similar Time Sequences Under Time Warping. In *Proceedings of the International Conference on Data Engineering*, pages 201–208, Orlando, Florida, February 1998. IEEE Computer Society.

[133] Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard T. Snodgrass, V. S. Subrahmanian, and Roverto Zicari. *Advanced Database Systems.* Data Management Systems. Morgan Kaufmann, San Francisco, CA, 1997.

[134] Donghui Zhang. *Aggregation Computation over Complex Objects.* PhD thesis, University of California, Riverside, August 2002.

[135] Donghui Zhang, Dimitris Gunopulos, Vassilis J. Tsotras, and Bernhard Seeger. Temporal Aggregation over Data Streams using Multiple Granularities. In *Proceedings of the Conference on Extending Database Technology*, pages 646–663, Prague, Czhech Republic, March 2002.

[136] Donghui Zhang, Alexander Markowetz, Vassilis J. Tsotras, Dimitrios Gunopulos, and Bernhard Seeger. Efficient Computation of Temporal Aggregates with Range Predicates. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 237–245, Santa Barbara, CA, May 2001.

[137] Yunyue Zhu and Dennis Shasha. Warping Indexes With Envelope Transforms for Query by Humming. In *Proceedings of the ACM-SIGMOD Conference*, pages 181–192, San Diego, CA, June 2003.