UNIVERSITY OF CALIFORNIA SAN DIEGO

Accelerating Analytic Queries on Compressed Data

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Chunbin Lin

Committee in charge:

Professor Yannis Papakonstantinou, Chair
Professor Alin Deutsch
Professor Tara Javidi
Professor Arun Kumar
Professor Victor Vianu

2018

The Dissertation of Chunbin Lin is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2018

DEDICATION

To my family and advisor, I could not have done it without you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

During the past five years as a Ph.D. student, I am so blessed to meet so many great people – advisors, colleagues, friends, and family, without whom this dissertation cannot be finished.

First, I was honored and fortunate to spend five years with my Ph.D. advisor Professor Yannis Papakonstantinou, who gave me the opportunity to conduct the research in an extremely free environment. I dedicate my deepest gratitude for his guidance, encouragement, thoughts, and supports that will remain in my heart forever.

Besides my advisor, I would also like to thank my committee members, Professor Alin Deutsch, Professor Tara Javidi, Professor Arun Kumar, and Professor Victor Vianu, for their constructive feedback/comments that helped complete my thesis and dissertation.

I also would like to thank Dr. Michalis Petropoulos who was the manager of my internship in Amazon during the summer of 2017, Dr. Ippokratis Pandis and Dr. Fabian Nagel who were the mentors of my internship in Amazon during the summer of 2017, and Mr. Mohan Sankaran who was the manager of my internship in Informatica during the summer of 2014 and 2015. With the help from the managers and mentors, I gained valuable experiences and enjoyed unforgettable summers.

I am blessed to be surrounded by so many good friends and colleagues who made University of California San Diego a truly unforgettable memory in my life. I thank my colleagues and friends for the insightful discussions and all the fun. Particularly, I would like to thank Xun Jiao, Yanqin Jin, Jing Li, Yuliang Li, Jianguo Wang, Jiapeng Zhang, and Costas Zarifis. Life is partly what we make it and partly what it is made by the friends we choose.

Finally, I would not be able to achieve anything without the support and understanding from my beloved wife Wanxia Zhong and my daughter Rongxing (Lucy) Lin.

The material in this dissertation is based on the following publications.

Chapter 2 contains material from "Fast In-Memory SQL Analytics on Typed Graphs" by Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou and Matthias Springer, which appears

in Proceedings of VLDB Endowment, Volume 10, Number 3, November 2016. The dissertation author was the primary investigator of this paper.

Chapter 3 contains material from "Plato: Approximate Queries over Compressed Time Series with Tight Deterministic Error Guarantees" by Etienne Boursier, Jaqueline J. Brito, Chunbin Lin, and Yannis Papakonstantino, which was submitted for publication, and "Supporting Anytime Queries over Compressed Time Series with Deterministic Error Guarantees" by Chunbin Lin, Joshua Lapacik, and Yannis Papakonstantino, which is in preparation for submission. The dissertation author was the primary investigator of the papers.

My coauthors (Benjamin Mandel, Matthias Springer, Etienne Boursier, Jaqueline J. Brito, Joshua Lapacik, and Professor Yannis Papakonstantinou) have all kindly approved the inclusion of the aforementioned publications in my dissertation.

VITA

2010	B.Sc., Management Information Systems, Renmin University of China
2013	M.Sc., Computer Science, Renmin University of China
2018	Ph.D., Computer Science and Engineering, University of California, San Diego

PUBLICATIONS

- Optimal algorithms for selecting top-k combinations of attributes: theory and applications. The International Journal on Very Large Data Bases (VLDB Journal), 27(1), pp. 27-52, 2018.
- GQFast: Fast Graph Exploration with Context-aware Autocompletion. IEEE International Conference on Data Engineering (ICDE), pp. 1389-1390, 2017.
- MILC: inverted list compression in memory. Proceedings of the VLDB Endowment (PVLDB), 10(8), pp. 853-864, 2017.
- Fast and Scalable Distributed Set Similarity Joins for Big Data Analytics. IEEE International Conference on Data Engineering (ICDE), pp. 1059-1070, 2017.
- An experimental study of bitmap compression vs. inverted list compression. ACM International Conference on Management of Data (SIGMOD), pp. 993-1008, 2017.
- HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. Proceedings of the VLDB Endowment (PVLDB), 9(14), pp. 1647-1658, 2016.
- Fast In-Memory SQL Analytics on Typed Graphs. Proceedings of the VLDB Endowment (PVLDB), 10(3), pp. 265-276, 2016.
- Sherlock: Sparse Hierarchical Embeddings for Visually-Aware One-Class Collaborative Filtering. International Joint Conferences on Artificial Intelligence (IJCAI), pp. 3740-3746, 2016.
- Boosting the quality of approximate string matching by synonyms. ACM Transactions on Database Systems (TODS), 40(3), 2015.
- String similarity measures and joins with synonyms. ACM International Conference on Management of Data (SIGMOD), pp. 373-384, 2013.
- LotusX: A Position-Aware XML Graphical Search System with Auto-Completion. IEEE International Conference on Data Engineering (ICDE), pp. 1265-1268, 2012.
- Optimal top-k generation of attribute combinations based on ranked lists. ACM International Conference on Management of Data (SIGMOD), pp. 409-420, 2012.

ABSTRACT OF THE DISSERTATION

Accelerating Analytic Queries on Compressed Data

by

Chunbin Lin

Doctor of Philosophy in Computer Science

University of California San Diego, 2018

Professor Yannis Papakonstantinou, Chair

Data compression techniques (both lossless and lossy compression methods) are widely utilized in big data analytic applications in domains including health-care, transportation, and finance. The main benefit achieved from applying data compression techniques is the saving of space cost. However, performing analytic queries on compressed data has two major challenges in terms of the performance and the accuracy: (i) decompressing data may damage the performance, and (ii) if lossy data compression techniques are utilized then the returned answers are not accurate. In this dissertation, we study how to accelerate analytic queries over compressed data (and provide tight error guarantees for approximate answers if lossy data compression methods are applied).

First, this dissertation introduces an in-memory database, called *GQ-Fast*, which supports high-performance analytic SQL queries over compressed relational data. We study a class of graph analytic SQL queries, called relationship queries. These queries involving aggregation, join, semijoin, intersection and selection are a wide superset of fixed-length graph reachability queries and of tree pattern queries. We present real-world OLAP scenarios, where efficient relationship queries are needed. However, row stores, column stores and graph databases are unacceptably slow in such OLAP scenarios. To support efficient relationship queries, we propose a GQ-Fast database, which is an indexed database that roughly corresponds to efficient encoding of annotated adjacency lists that combines salient features of column-based organization, indexing and compression. GQ-Fast uses a bottom-up fully pipelined query execution model, which enables (i) aggressive compression (e.g., compressed bitmaps and Huffman) and (ii) avoids intermediate results that consist of row IDs (which are typical in column databases). In addition, GQ-Fast compiles query plans into executable C++ source code. Besides achieving runtime efficiency, GQ-Fast also reduces main memory requirements because, unlike column databases, GQ-Fast selectively allows dense forms of compression including heavy-weight compressions, which do not support random access. GQ-Fast outperforms the state-of-the-art databases by 1-3 orders of magnitudes and GQ-Fast uses less space.

Second, this dissertation proposes an approximate query processing system, called *Plato*, which supports anytime analytic queries over compressed time series with *sound and tight deterministic error guarantees*. Plato supports expressions that are compositions of the linear algebra operators over vectors along with arithmetic operators. Such analytics can express common statistics (such as correlation and cross-correlation) that may combine multiple time series. Plato builds a compressed segment tree structure for each time series during the offline insertion time. Each node in the tree refers to a time series segment. Each segment (i) is compressed by an estimation function that approximates the actual values and is coming from a user-chosen estimation function family, and (ii) is associated with one to three (depending on the case) precomputed error measures. Then Plato is able to provide tight deterministic error

guarantees satisfying the error budgets provided by users by accessing the minimal number of nodes in the segment tree. In addition, we also identify two broad estimation function family groups. The *Vector Space (VS)* family and the presently defined *Linear Scalable Family (LSF)* lead to theoretically and practically high-quality guarantees, even for queries that combine multiple time series that have been independently compressed. Well-known function families (e.g., the polynomial function family) belong to LSF. The theoretical aspect of "high quality" is crisply captured by the *Amplitude Independence (AI)* property: An AI guarantee does not depend on the amplitude of the involved time series, even when we combine multiple time series.

Chapter 1

Introduction

Systems supporting efficient big data analytics are highly required in many domains ranging including healthcare, transportation, and finance [87, 86, 42, 12]. However, the size of data keeps increasing fast. Developing such systems is challenging due to the large size of the data. In addition, the size of data keeps increasing. For example, an IoT-ready oil drilling rig produces about 8 TB of operational data in one day. [1]

To reduce the size of data, data compression techniques (both lossless and lossy compression methods) are widely utilized in big data analytic applications. For example, relational databases usually adopt the lossless compression methods to compress the relational data, e.g., Run-length Encoding (RLE) [91, 62], Bitmap [100, 63] and Dictionary encoding [22, 62], while time series databases lossy compress time series with functions, e.g, Piecewise Linear Representation (PLR) [43], Fourier Transforms [10], and Wavelets [81].

Although applying data compression methods can significantly reduce the size of data, it brings challenges in answering analytic queries over compressed data in terms of the performance and the accuracy: (i) Decompressing compressed data may be time-consuming, which may dominate the overall running time. For example, decompressing (ii) If lossy data compression techniques are utilized then the returned answers are approximate instead of accurate. It is necessary to provide error guarantees for approximate answers. In this dissertation, we study how to accelerate analytic queries over compressed data (and provide tight error guarantees for

[1] https://wasabi.com/storage-solutions/internet-of-things/

approximate answers if lossy data compression methods are applied).

We first introduce an in-memory indexed database, called *GQ-Fast* [63], which supports high-performance analytic SQL queries over compressed relational data in Chapter 2. Then we present an approximate query processing system called *Plato*, which supports efficient anytime analytic queries over compressed time series with sound and tight deterministic error guarantees in Chapter 3. Notice that GQ-Fast adopts lossless data compression methods to compress relational data, while Plato uses lossy data compression techniques to compress time series.

GQ-Fast. We study a class of graph analytics SQL queries, which we call *relationship queries*. These queries involving aggregation, join, semijoin, intersection and selection are a wide superset of fixed-length graph reachability queries and of tree pattern queries. We present real-world OLAP scenarios, where efficient relationship queries are needed. However, row stores, column stores and graph databases are unacceptably slow in such OLAP scenarios.

To support efficient relationship queries, we propose the GQ-Fast database, which is an indexed database that roughly corresponds to efficient encoding of annotated adjacency lists that combines salient features of column-based organization, indexing and compression. GQ-Fast uses a bottom-up fully pipelined query execution model, which enables (a) aggressive compression (e.g., compressed bitmaps and Huffman) and (b) avoids intermediate results that consist of row IDs (which are typical in column databases). GQ-Fast compiles query plans into executable C++ source code. Besides achieving runtime efficiency, GQ-Fast also reduces main memory requirements because, unlike column databases, GQ-Fast selectively allows dense forms of compression including heavy-weight compressions, which do not support random access.

We used GQ-Fast to accelerate queries for two OLAP dashboards in the biomedical field. GQ-Fast outperforms PostgreSQL by $2-4$ orders of magnitude and MonetDB, Vertica and Neo4j by $1-3$ orders of magnitude when all of them are running on RAM. Our experiments dissect GQ-Fasts advantage between (i) the use of the compiled code, (ii) the bottom-up pipelining execution strategy, and (iii) the use of dense structures. Other analyses and experiments show the space savings of GQ-Fast due to the appropriate use of compression methods. We also show

that the runtime penalty incurred by the dense compression methods decreases as the number of CPU cores increases.

Plato. We provide an approximate query processing system, called *Plato*, which provides sound and tight deterministic error guarantees for anytime analytic queries over compressed time series. Plato supports expressions that are compositions of the (commonly used in time series analytics) linear algebra operators over vectors, along with arithmetic operators. Such analytics can express common statistics (such as correlation and cross-correlation) that may combine multiple time series.

For each time series, Plato builds a compressed segment tree structure. Each node in the tree corresponds to a time series segment. Each segment (i) is compressed by an estimation function that approximates the actual values and is coming from a user-chosen estimation function family, and (ii) is associated with one to three (depending on the case) precomputed error measures. Then Plato is able to provide tight deterministic error guarantees over the compressed time series. Plato allows users to input expressions and error budgets, then Plato accesses the minimal number of nodes in the corresponding segment trees to output approximate answers with tight deterministic error guarantees, which are no greater than the given error budgets.

In addition, we identify two broad estimation function family groups. The *Vector Space (VS)* family and the presently defined *Linear Scalable Family (LSF)* lead to theoretically and practically high-quality guarantees, even for queries that combine multiple time series that have been independently compressed. Well-known function families (e.g., the polynomial function family) belong to LSF. The theoretical aspect of "high quality" is crisply captured by the *Amplitude Independence (AI)* property: An AI guarantee does not depend on the amplitude of the involved time series, even when we combine multiple time series. The experiments on four real-life datasets validated the importance of the Amplitude Independent (AI) error guarantees: When the novel AI guarantees were applicable, the guarantees could ensure that the approximate query results were very close (typically 1%) to the true results.

Dissertation organization. In Chapter 2 we introduce the GQ-Fast system which supports SQL queries over compressed relational data. We introduce the Plato system supporting anytime analytic queries over compressed time series in Chapter 3. Finally, Chapter 4 concludes the dissertation and gives future directions.

Chapter 2

Fast In-Memory SQL Analytics on Typed Graphs

2.1 Introduction

The focus of past OLAP systems was on SQL queries on data cubes, whose data is modeled as star/snowflake SQL schemas [16, 97]. However, in recent years, an avalanche of graph data emerged, such as disease-drug networks (chem/bio-informatics) [50, 51] and social networks (Web) [103]. A new generation of benchmarks, such as the Microsoft Academic Graph (MAG) Benchmark [93] and the Berkeley Big Data Benchmark [80] make clear the distinction of these data from data cubes (such as the old TPC-H benchmark). The particular data sets and benchmarks, as well as many others, are essentially *typed graphs*, i.e., graphs where vertices and edges are associated with types known in advance. There is an increasing demand to perform analytic SQL queries over such graphs; e.g., discovering related diseases in a disease-drug network graph. Traditional, SQL OLAP technologies do not handle such demands well because they are not sufficiently optimized for finding paths among entities [19, 18].

Schema. Towards SQL-based OLAP on graphs, we first define the representation of *typed graphs* (also known as graphs with schema, e.g., [39]) in an SQL database. The nodes and edges of a typed graph are represented as tuples of relational tables. We classify the tables into two categories: *Entity tables* and *Relationship tables*, following the database E/R model [32]. We focus on *binary* relationships. Each entity table has a primary key column, called the *ID* column,

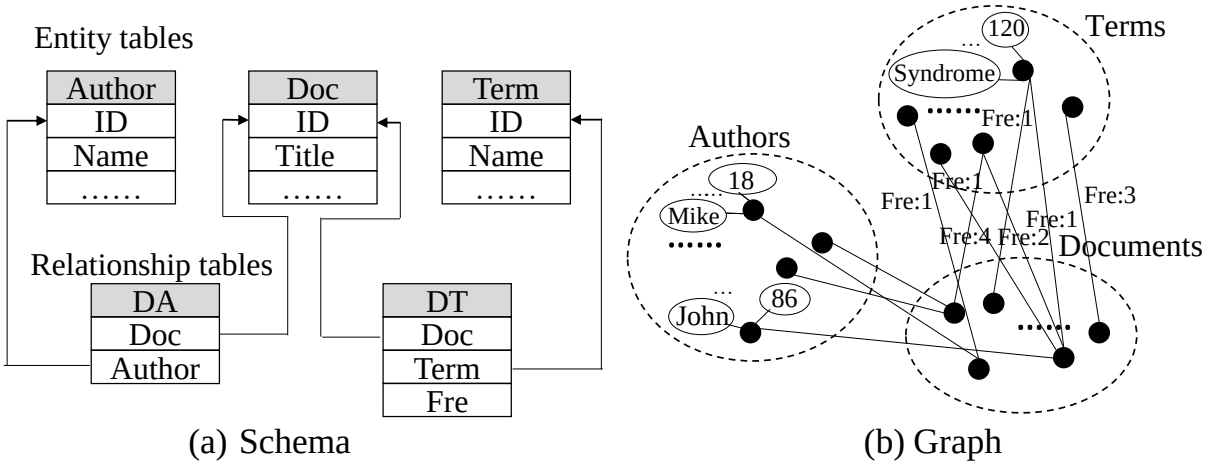


Figure 2.1. PubMed schema and the corresponding graph model.

while each relationship table has two foreign key columns pointing to ID columns of entity tables[1]. Hence, the tuples comprise a typed graph [101, 105]: Each entity table corresponds to a type of vertices, while each relationship table corresponds to a type of edges. Columns in entity tables and relationship tables correspond to attributes of vertices and edges, respectively. For example, consider the premier public biomedical database PubMed[2]. Figure 2.1(a) shows its schema, and Figure 2.1(b) presents a corresponding typed graph. Each entity table corresponds to a type of vertices, while each relationship table corresponds to edges linking corresponding types of vertices. The tuples of the relationship table DT stand for edges from a document tuple/entity/node to a term tuple/entity/node.

Relationship Queries. We identify a class of queries, called *relationship queries*, which cover many analytics needs on graph data and, in addition, they are amenable to orders-of-magnitude speed optimization. Informally, a relationship query contains three steps.

- *Context Computation*: The context is a collection of entities whose properties satisfy the user given conditions.

- *Path Navigation*: Navigating from source entities to target entities is via joins over relationship

[1]In order to capture many-to-one relationships efficiently, we also allow entity tables to have foreign keys [32]. We neglect this possibility, as it does not essentially change any consideration.

[2]http://www.ncbi.nlm.nih.gov/pubmed

tables.

- *Path Aggregation*: The importance of the target entities is computed by applying aggregation functions over attributes collected along the navigation paths.

Notice that the first and third steps are optional. Relationship queries are common in graph analytics. For example, all the queries evaluated in [65] are relationship queries. We illustrate a relationship query on the PubMed schema, which will serve as one of the running examples.

Query SD (Similar Documents). Assume a user wants to find documents d_j that are similar to a given document d_0 with ID d_0^{ID} in the PubMed graph. Similarity between documents d_0 and d_j is measured by the number of terms associated to both of them, i.e., the number of paths with type **Doc** $\rightarrow$ **Term** $\rightarrow$ **Doc** that start at d_0 and end at d_j. The corresponding SQL query is shown below.

```
SELECT dt2.Doc, COUNT(⋆) AS similarity
FROM DT dt1 JOIN DT dt2 ON dt1.Term = dt2.Term
WHERE dt1.Doc = d_0^ID
GROUP BY dt2.Doc
```

The Query SD is a simple relationship query: It navigates via typed paths **Doc** $\rightarrow$ **Term** $\rightarrow$ **Doc** and then aggregates the number of paths reaching each target. More complex (and performance-challenging) relationship queries are presented in Chapter 2.2.

It is challenging to answer even this simple query efficiently due to the large size of the graph: thirty million vertices and one billion edges with several attributes. Given the analytical nature of relationship queries, column-oriented database systems are much more efficient than row-stores and graph database systems, as our experiments verified (see Chapter 2.6) [94, 3, 2, 40]. Nevertheless, the obtained performance is often insufficient for online queries and interactive applications. Query SD takes 61.6 and 19.17 seconds on the column databases MonetDB [40]

and Vertica [58], 741.2 seconds on the row database PostgreSQL, and 49.3 seconds on the graph database Neo4j, even though we fully cached the data in main memory in all the cases. Performance gets far worse when the join paths are longer, the aggregations involve many attributes of the paths or the source entities themselves are specified by their properties and connections, rather than their IDs.

To improve the performance, we propose an index-only fully pipelined database called *GQ-Fast*. GQ-Fast answers Query SD in 1.068 seconds. As the queries become more complex, its performance ratio to the other systems widens. Moreover, GQ-Fast generally requires less memory.

GQ-Fast achieves such superior performance by employing a code generator to produce efficient fully pipelined source code running upon a new compressed fragment-based index, as outlined in the following paragraphs.

Database Structure. A GQ-Fast database physically stores only indices – it does not store the logical tables. Generally, the administrator may load a relation $R(C_1, C_2, ..., C_n)$ and specify that for each ID or foreign key attribute C a respective index should be built, using C as the *indexed column*. In response, GQ-Fast will make an index $\mathcal{I}_{R.C}$ for each such attribute. Figure 2.2 shows two indices $\mathcal{I}_{\texttt{DT.Doc}}$ and $\mathcal{I}_{\texttt{DT.Term}}$ that correspond to the two foreign keys of the table **DT**. During runtime, the GQ-Fast query processor will use the index to find (projections of) tuples of R that have a given $C = c$ value. For example, the index $\mathcal{I}_{\texttt{DT.Doc}}$ can be used to find the terms associated to document 116 (i.e., $\pi_{\texttt{Term}}\sigma_{\texttt{Doc}=116}\mathbf{DT}$) or to find the term/frequency pairs associated to document 116 (i.e., $\pi_{\texttt{Term, Fre}}\sigma_{\texttt{Doc}=116}\mathbf{DT}$).

Internally, a GQ-Fast index has two components: a lookup table and a set of fragments. Let us say, w.l.o.g., that C_1 is the indexed column. Then for each column $C_j \in \{C_2, \ldots, C_n\}$ and for each value $t \in C_1$ there is a fragment $\pi_{C_j}\sigma_{C_1=t}(R)$, which retains the original order of the values. In Figure 2.2, the fragment $\pi_{\texttt{Term}}\sigma_{\texttt{Doc}=116}\mathbf{DT}$ (with contents 28, 66, etc.) and the fragment $\pi_{\texttt{Fre}}\sigma_{\texttt{Doc}=116}\mathbf{DT}$ (with contents 6, 3, etc.) provide the terms and frequencies associated to document 116, respectively.

To reduce space costs, GQ-Fast compresses individual fragments. *The key observation behind compressing fragments is that when a relationship query accesses a fragment, all of its data will be used. There is no need for random access within the fragment.* Based on this, GQ-Fast allows very compressed encodings of each individual fragment, such as Huffman encoding. Note that the typical fragment is relatively small (compared to the column) and can typically fit in the L1 cache or, at least, in the L2 cache. Hence, its decoding is not penalized with multiple random access to the RAM.

Given a value for the indexed column, the lookup table must be able to provide a pointer to the respective fragment, along with the size of each fragment. A lookup table can be built in many known ways; e.g., as a hash table. GQ-Fast saves space and response time by building *lookup tables as offset arrays that utilize the dense ID assumption*, according to which the IDs of an entity table are consecutive integers, starting from 0. Under this assumption, all fragments of the same type are listed consecutively in an array: A GQ-Fast lookup table for an index on $\texttt{R.C}_1$ is a two-dimensional array $\mathcal{I}_{\texttt{R.C}_1}$ of size $v \times (n-1)$, where v is the number of unique values in $\texttt{R.C}_1$ and n is the number of columns in R. The starting address of the fragment $\pi_{C_j}\sigma_{C_1=t}(R)$ is stored in $\mathcal{I}_{\texttt{R.C}_1}[t][j-1]$ and its size can be calculated using the starting address of the next fragment.

Query Processing. GQ-Fast query plans run exclusively on indices. They employ a *bottom-up pipelined execution model*, illustrated next, to avoid large intermediate results. In addition, GQ-Fast employs a C++ code generator for query plans.

As an example, consider the generated code for Query SD, which uses table **DT** with columns **Document** (0^{th} column) and **Term** (1^{st} column). In Lines 2–4, GQ-Fast uses index $\mathcal{I}_{\texttt{DT.Doc}}$ to find the Terms' fragment $\pi_{\texttt{Term}}\sigma_{\texttt{Doc}=116}$ ($\texttt{DT} \mapsto \texttt{dt1}$) starting at position $\mathcal{I}_{\texttt{DT.Doc}}[116][1]$ with size $l_{\texttt{dt1.Term}}$. GQ-Fast decodes the fragment into the (preallocated) array $\mathcal{A}_{\texttt{dt1.Term}}$ and returns the number of elements $n_{\texttt{dt1.Term}}$. Afterwards (Lines 5–9), for each term ID $v_{\texttt{dt1.Term}} \in \mathcal{A}_{\texttt{dt1.Term}}$, it uses index $\mathcal{I}_{\texttt{DT.Term}}$ to find fragments $\pi_{\texttt{dt2.Doc}}\sigma_{\texttt{dt2.Term}=v_{\texttt{dt1.Term}}}$ ($\texttt{DT} \mapsto$ $\texttt{dt2}$) starting at position offset $\mathcal{I}_{\texttt{DT.Term}}[v_{\texttt{dt1.Term}}][0]$ (see Figure 2.2). GQ-Fast decodes the

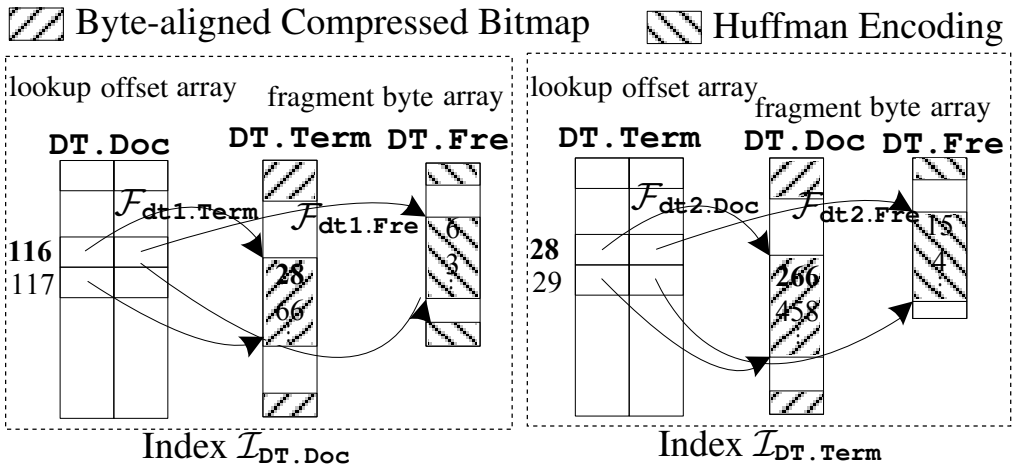


Figure 2.2. Example of fragments and query processing for query SD. Fragments $\pi_{Term}\sigma_{Doc=116}DT$ and $\pi_{Fre}\sigma_{Doc=116}DT$ are encoded with *bit-aligned compressed array* and *Huffman encoding*.

Generated Code: Generated code for Query SD

```
1  R ← ∅
2  F_dt1.Term ← I_DT.Doc[116][1]
3  l_dt1.Term ← I_DT.Doc[116 + 1][1] − F_dt1.Term
4  A_dt1.Term, n_dt1.Term ← decodeBB(F_dt1.Term, l_dt1.Term)
5  for i ← 0 to n_dt1.Term − 1 do
6      v_dt1.Term ← A_dt1.Term[i]
7      F_dt2.Doc ← I_DT.Term[v_dt1.Term][0]
8      l_dt2.Doc ← I_DT.Term[v_dt1.Term + 1][0] − F_dt2.Doc
9      A_dt2.Doc, n_dt2.Doc ← decodeBB(F_dt2.Doc, l_dt2.Doc)
10     for j ← 0 to n_dt2.Doc − 1 do
11         v_dt2.Doc ← A_dt2.Doc[j]
12         R[v_dt2.Doc] ← R[v_dt2.Doc] + 1
13 return R
```

identified fragments into $\mathcal{A}_{\texttt{dt2.Doc}}$. Finally, GQ-Fast scans all Documents fragments to update the array $\mathcal{R}$ (Lines 11–12), which holds the counts per document.

Notice that (1) GQ-Fast can afford to have $\mathcal{R}$ be an array (as opposed to a hash table) because of the dense ID assumption; (2) The execution is pipelined in a sense that it iterates over the fragments and their elements. The memory footprint is small as it is dictated by the max. size of fragments and not of the overall column size.

Contributions. This paper makes the following contributions.

- Formally identifies *relationship queries*, a subset of SQL that is both important and amenable to orders-of-magnitude optimization. We illustrate the subset's importance with (a) examples from two real-world use cases (Chapter 2.2) and (b) by providing their syntax, showing that it captures a very large part of SQL.

- Describes a new *index-only and fragment-based data organization* (Chapter 2.4) and coordinated query plans (Chapter 2.5). The bottom-up query plan enables (a) aggressive compression (e.g., compressed bitmaps and Huffman) and (b) avoids intermediate results that consist of row IDs (which are typical in column databases). Further space savings and speed improvements stem from using a *dense* (consecutive) entity IDs assumption.

- Describes a code generator that produces C++ code for each query plan. The produced code benefits from CPU-level optimizations (Chapter 2.5).

- Performs experiments on three real-life datasets (i.e., PubMed-M, PubMed-MS [50] and SemMedDB [51]), showing GQ-Fast is $10 - 10^4$ times more efficient than MonetDB, Neo4j, Vertica and PostgreSQL when all are running on main memory (Chapter 2.6). Since the performance advantage is due to many factors, a comprehensive series of experiments isolates the marginal effect of each individual factor.

GQ-Fast has been deployed in two real world use cases around PubMed and SemMedDB data, has been released online [64].

2.2 Data Schema and Queries

2.2.1 Data Schema

We classify relational tables in GQ-Fast in two categories according to the entities and the relationships of the E/R model [32]: *entity tables* (e.g., **`Author`** in Figure 2.1(a)) and

relationship tables (e.g., **DT**, **DA**). Each entity table $\mathbb{E}$ has an ID (primary key) attribute and several attributes $M_1, ..., M_n$. Each tuple $t \in \mathbb{E}$ corresponds to a real-life entity. A relationship table $\mathbb{R}$ has two *foreign key attributes* F_1 and F_2 referencing the IDs of respective entity tables[3], i.e., $F_1 \rightsquigarrow \mathbb{E}_1.ID$ and $F_2 \rightsquigarrow \mathbb{E}_2.ID$, where $\rightsquigarrow$ is *reference*. The combination $(f_1, f_2) \in F_1 \times F_2$ is unique. A relationship table may also have *measure attributes* $M_1, ..., M_m$ (e.g., **DT.Fre**).

E/R schema $\rightarrow$ Graph model. Mapping E/R schemas to graph models is a well-studied topic [101, 11]. We use the following two steps to map our schema to a typed graph [39]: (i) Each entity table $\mathbb{E}(M_1, ..., M_n)$ refers to a type of vertices $\mathbb{V}$, and each entity $t \in \mathbb{E}$ refers to one vertex $v \in \mathbb{V}$. The attributes $M_1, ..., M_n$ in the entity table are mapped to properties of vertices; and (ii) each relationship table $\mathbb{R}(F_1, F_2, M_1, ..., M_m)$ refers to edges $\mathbb{E}$ crossing two types of vertices $\mathbb{V}_1 \times \mathbb{V}_2$, where $\mathbb{V}_1$ and $\mathbb{V}_2$ are translated from entities $\mathbb{E}_1$ and $\mathbb{E}_2$ and $F_1 \rightsquigarrow \mathbb{E}_1.ID$ and $F_2 \rightsquigarrow \mathbb{E}_2.ID$.

Graph model$\rightarrow$ E/R schema. Mapping a graph to a relational schema has been studied for several years [21, 104]. We first show how to convert a typed graph to our E/R schema, then describe general graphs. Mapping a typed graph to our schema has the following two steps: First, store vertices of the same type into one entity table. Each attribute of the vertices becomes one column in the table. Second, store edges that have the same type into the same relationship table. Edges of the same type have source (resp. target) nodes that have the same type.

For general graphs, the basic way is to store all the vertices in one big *Node(ID, Type)* table, while all the edges are in one *Edge(Source, Destination, Type)* table. If more detailed knowledge about the types of vertices can be inferred from the graph, then the mapping approach of typed graphs can be more fine-grained.

2.2.2 Relationship Query

Informally, a relationship query proceeds in three steps: (i) *Context Selection:* Entities satisfying query conditions (i.e., certain user-provided properties) are marked as a context; (ii)

[3]In this paper, we focus on relationship tables with two foreign keys.

Path Navigation: To reach target entities from the context, queries "navigate" between entities via join operations; and (iii) *Relevance Computation:* The relevance between each target entity and the context is computed by applying aggregation functions over measure attributes collected in the second step.

In its algebraic form, a relationship query involves σ (selection), π (projection), $\bowtie$ (join), $\ltimes$ (semi-join) operators and an optional γ (aggregation) at the end, and must satisfy the follow restrictions: (i) join and semijoin conditions are equalities between (primary or foreign) key attributes and (ii) aggregations group-by on a primary key or foreign key. The set of relationship queries includes graph reachability (path finding) queries, where the edges are defined by foreign keys. More generally, it includes tree pattern queries, followed by aggregation. The first restriction does not narrow down the scope of relationship query applications as it only requires that navigation on a graph should be performed via connected edges, which is a natural requirement for graph navigation. The second restriction allows GQ-Fast to use an array to maintain the aggregation results instead of using a map, which contributes to 30% performance improvement (see Table 2.9 in Chapter 2.6.2.3).

Example Queries. We now illustrate a number of relationship queries using the datasets of some GQ-Fast applications: PubMed and SemMedDB. These queries were used in our experiments and are implemented in our interactive demo system[4].

Even though the definition of relationship queries includes a larger set of queries, we focus on these queries, because they illustrate accurately the use cases for which GQ-Fast was designed and achieves the best speedup compared to other database systems: queries with long join paths involving many-to-many relationships. In the following examples, we use $E_1 \rightarrow E_2$ to visualize a join from table E_1 to table E_2 and $\circlearrowleft_E$ to visualize an intersection on table E.

[4]http://chunbinlin.com/demoGQFast/

2.2.2.1 Queries in PubMed

FSD (Frequency-Time-aware Document Similarity). Query FSD computes time-aware and frequency-aware cosine similarity. The cosine similarity is computed as follows: Each document d is associated with a vector $t^d = [t_1^d, \ldots, t_n^d]$, where n is the number of terms across all documents. The cosine similarity between two documents x and y is defined as $\sum_{i=1,\ldots,n} t_i^x t_i^y$[5] In contrast to Query SD in the Introduction, Query FSD raises the similarity degree of documents that are chronologically close. The navigation path of Query FSD can be visualized as **d1** $\rightarrow$ **dt1** $\rightarrow$ **dt2** $\rightarrow$ **d2**. The corresponding SQL query is shown below.

```
SELECT dt2.Doc, SUM(dt1.Fre * dt2.Fre) / abs(d1.Year-d2.Year)+1
FROM (((Doc d1 JOIN DT dt1 ON d1.ID = dt1.Doc)
        JOIN DT dt2 ON dt1.Term = dt2.Term)
       JOIN Doc d2 ON d2.ID = dt2.Doc)
WHERE d1.ID = d_0^ID
GROUP BY dt2.Doc
```

AD (Authors' Discovery). Query AD finds the authors who published papers that pertain to the terms identified by $t_1^{ID}, \ldots, t_n^{ID}$ (e.g., authors that published papers related to the terms "neoplasms" and "statins") and counts the number of papers per author. The navigation path of Query AD can be visualized as $\circlearrowleft_{dt} \rightarrow$ **da**.

```
SELECT da.Author, COUNT(*)
FROM DA da
WHERE da.Doc IN
  (SELECT dt.Doc FROM DT dt WHERE dt.Term = t_1^ID)
  INTERSECT
  ...
```

[5] In practice, the queries also normalize for the sizes of t^x and t^y and, in later examples, the sizes of measures. The examples exclude the normalization since they do not present any important additional aspect to the exhibited query pattern.

```
  INTERSECT
  (SELECT dt.Doc FROM DT dt WHERE dt.Term = t_n^ID)
GROUP BY da.Author
```

Note that relationship queries do not require that all subqueries have identical structure. Furthermore, the aggregation is not necessary. For example, the following query finds authors who have recently (after 2012) published a paper on "statins" (term id 583352) and at least one of the paper's authors had also published on "lung neoplasms" (term id 384053).

```
SELECT da.Author
FROM DA da
WHERE da.Doc IN
   (SELECT dt.Doc FROM DT dt WHERE dt.Term = 583352)
   INTERSECT
   (SELECT d.ID FROM Document d WHERE d.Year > 2012)
   INTERSECT
   (SELECT da.Doc FROM DA da JOIN DT dt ON da.Doc = dt.Doc
     WHERE dt.Term = 384053)
```

FAD (Co-Occurring Terms Discovery). Query FAD is similar to Query AD. It finds other terms that co-occur in documents about terms identified by $t_1^{ID}, \ldots, t_n^{ID}$ along with the number of occurrences (e.g., terms that co-occur in documents about "neoplasms" and "statins" and how often). The navigation path of Query FAD can be visualized as $\circlearrowleft_{dt} \rightarrow$ **dt1**.

```
SELECT dt1.Term, Sum(dt0.Fre)
FROM DT dt1
WHERE dt.Doc IN
  (SELECT dt.Doc FROM DT dt1 WHERE dt1.Term = t_1^ID)
  INTERSECT
```

```
  ...
  INTERSECT
  (SELECT dt.Doc FROM DT dtn WHERE dtn.Term = t_n^ID)
GROUP BY dt1.Term
```

AS (Author Similarity). Query AS specifies an author with ID a^{ID} and finds other authors that have similar publications, weighing higher those with publications in recent years. The similarity is measured by the frequency-aware cosine of common terms between the publications. It also favors documents that are chronologically close. The navigation path of AS can be visualized as **da1** $\rightarrow$ **dt1** $\rightarrow$ **dt2** $\rightarrow$ **d** $\rightarrow$ **da2**. The SQL expression of AS is shown below.

```
SELECT da2.Author, SUM(dt1.Fre×dt2.Fre)/(2017-d.Year)
FROM (((DA da1 JOIN DT dt1 ON da1.Doc=dt1.Doc)
         JOIN DT dt2 ON dt1.Term = dt2.Term)
        JOIN Document d ON dt2.Doc=d.ID)
       JOIN DA da2 ON dt2.Doc=da2.Doc
WHERE da1.Author = a^ID
GROUP BY da2.ID
```

2.2.2.2 Queries in SemMedDB

The Scripps Research Institute implemented *Knowledge.Bio* , a system for exploring, learning, and hypothesizing relationships among concepts of the SemMedDB database[6], which is a repository of semantic predications (subject-predicate-object triples). Figure 2.3 shows the schema of SemMedDB.

CS (Concept Similarity). As a use case of Knowledge.Bio, Query CS finds the concepts that are most relevant to a given concept, e.g., "Atropine", where c^{ID} is the concept ID of "Atropine". The navigation path of Query CS can be visualized as **c1** $\rightarrow$ **p1** $\rightarrow$ **s1** $\rightarrow$ **s2** $\rightarrow$ **p2**

[6]http://skr3.nlm.nih.gov/SemMedDB/dbinfo.html

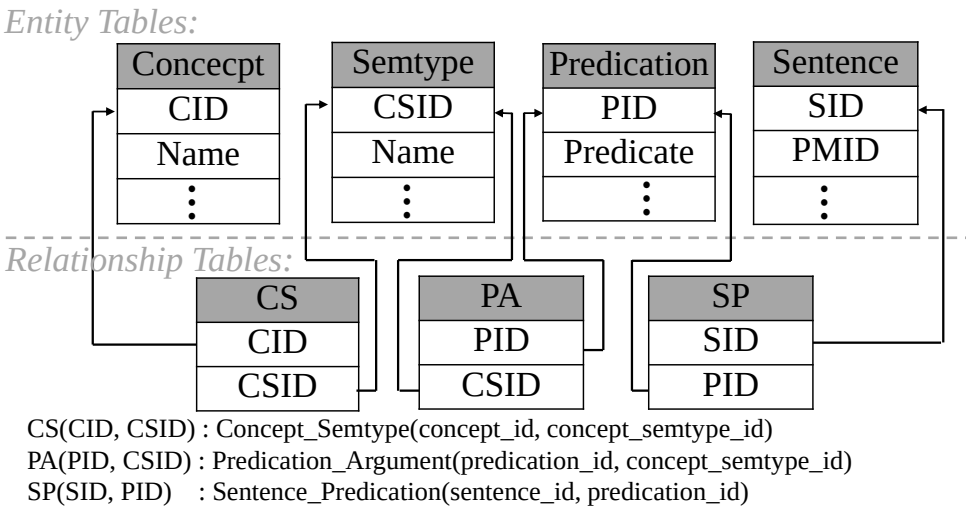


Figure 2.3. SemMedDB database schema. **CS**, **PA** and **SP** are relationship tables, whereas the others are entity tables.

$\rightarrow$ **c2**. The SQL expression of CS is shown below.

```
SELECT c2.CID, COUNT(*)
FROM CS c2, PA p2, SP s2
WHERE s2.PID = p2.PID
  AND p2.CSID = c2.CSID AND s2.SID IN
    (SELECT s1.SID
     FROM CS c1, PA p1, Sp s1
     WHERE s1.PID = p1.PID AND p1.CSID = c1.CSID
       AND c1.CID = c^ID)
GROUP BY CID
```

The running time of this query on an Amazon Relational Database Service (Amazon RDS) with MySQL was 25 minutes. GQ-Fast reduced the running time for that query to less than 1 second.

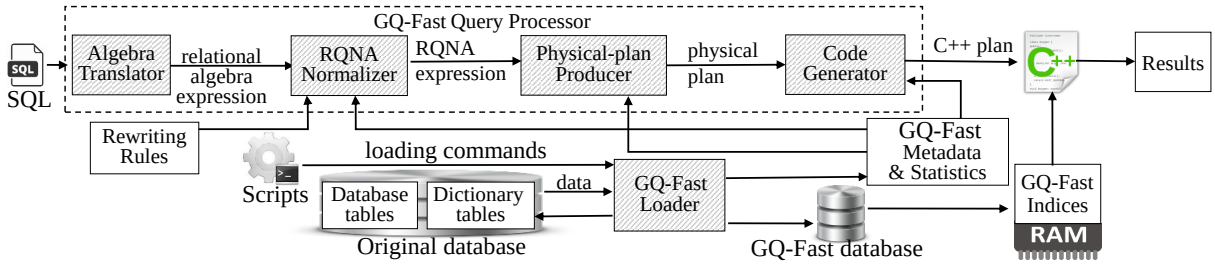


Figure 2.4. Architecture of GQ-Fast. The *GQ-Fast Loader* produces GQ-Fast databases and metadata and the *GQ-Fast Query Processor* generates code answering given relationship queries.

2.2.2.3 Further Examples

Relationship queries can be found in a variety of applications. Some examples are outlined in the following list.

Potential Virus Discovery in Network Security [30, 53]: Consider a database documenting virus infections in a computer network with tables for "virus" entities, "host IP" entities, and virus instance - host IP relationships. To discover potential virus infections for a host who has reported a virus s, a relationship query first selects the set of hosts associated with s, and then retrieves and aggregates all the virus infections know for these hosts. The viruses with the high scores might also hide in the host computer.

Friend Suggestion in Social Networks [90]: Consider a database with "user" entities, "tweet" entities, and a relationship associating tweets with users. For example, the relationship captures the information that a user read or shared a tweet. To provide friend suggestions for a given user u, a relationship query first discovers his/her tweets, then returns a sorted list of users based on their association with the discovered tweets.

2.3 Architecture

Applications use GQ-Fast as an OLAP-oriented database that accompanies their original transaction-oriented databases. Figure 2.4 gives an overview of GQ-Fast's architecture. It has two parts: *GQ-Fast Database Generation* and *GQ-Fast Query Processing*.

GQ-Fast Database Generation. The *GQ-Fast Loader* receives loading commands, retrieves data from one or multiple relational databases, and creates GQ-Fast indices along with relevant metadata, containing information about fragments and their encodings. This phase is done offline. The schema of the GQ-Fast database has to follow certain conventions (see Chapter 2.2). GQ-Fast data is stored in main memory data structures (see Chapter 2.4).

When loading data into GQ-Fast, users should specify (i) the columns to be indexed, upon which GQ-Fast builds lookup tables. Then, GQ-Fast organizes the values in other columns as fragments; and (ii) an encoding method for each column excluding indexed columns. Chapter 2.4 provides detailed guidelines for choosing proper encoding methods for different columns.

GQ-Fast Query Processing. The *GQ-Fast Query Processor* receives an SQL query and outputs its result. It consists of several subcomponents. The *Algebra Translator* translates an SQL query into a relational algebra expression, which is then transformed into a *Relationship Query Normalized Algebra (RQNA) expression* (see Chapter 2.5.1) by the *RQNA Normalizer* using rewriting rules. Given an SQL query q in its algebraic format, the RQNA Normalizer applies the following rewriting rules to transform it into RQNA: (1) push every possible selection and projection down to the corresponding tables; projections are upon selections, (2) rewrite into a *left-deep*.

The *RQNA Normalizer* also verifies whether an SQL query is a relationship query by checking the restrictions according to metadata. Afterwards, the *Physical-plan Producer* transforms the RQNA expression into a physical-level plan. The *Code Generator* consumes the physical plan and metadata and produces C++ code, which is then compiled and ran on the GQ-Fast index to get final results. GQ-Fast can also prepare a query statement, and then execute it multiple times (as JDBC does), changing the parameters each time.

We will illustrate GQ-Fast data structure and GQ-Fast query processing in Chapter 2.4 and Chapter 2.5 respectively.

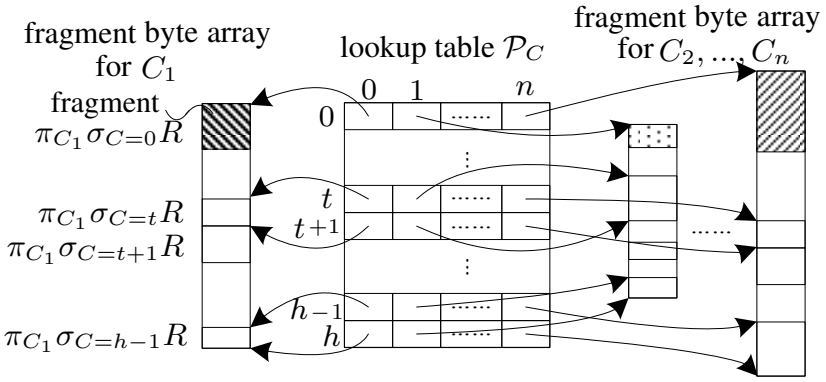


Figure 2.5. Index $\mathcal{I}_{R.C}$. The lookup table $\mathcal{P}_c$ stores offsets of fragments. Fragments of different columns have different encodings.

2.4 GQ-Fast Index Structure

This chapter first presents the GQ-Fast index structure and analyzes different encoding methods, then describes how to build indices for both entity and relationship tables. Finally, a discussion of how to support incremental updates is provided.

2.4.1 Index Structure

Given a relation $R(C, C_1, C_2, ..., C_n)$, assuming the indexed column is C, GQ-Fast builds one index $\mathcal{I}_{R.C}$ (shown in Figure 2.5) for R. A GQ-Fast index has one lookup table for the indexed column C, and organizes values in columns $C_1, ..., C_n$ in fragments. We assume that $|R| = h$, consequently the column C contains IDs in the interval $[0, h-1]$. The lookup table $\mathcal{P}_C$ is a 2D array of size $(h+1) \times n$ and stores offsets into the respective fragments array designating the beginning of a fragment. All fragments are stored consecutively and byte-aligned in one *fragment byte array* per column. Specifically, $\mathcal{P}[t][m]$ stores the offset where fragment $\pi_{C_m}\sigma_{C=t}(R)$ starts in C_m's fragment byte array, where C_m is the $(m+1)$-th column of R. If a value $t \in C$ has no associated values in columns $C_1, ..., C_n$, then all fragments $\pi_*\sigma_{C=t}(R)$ are empty. The size of a fragment is defined implicitly as the difference between two consecutive offsets, which is why the size of the first dimension of $\mathcal{P}$ is $h+1$. For further space savings,

offsets are encoded with the minimum number of bytes. For example, assume the C_1 fragment byte array (Figure 2.5) is 3GB and is located at a certain 64-bit memory address s. Offset values pointing to it will be 4-byte integers, since $\lceil \log_{256} 3 \times 1024 \times 1024 \times 1024 \rceil = 4$. Also, assume the offset to the t-th fragment $\pi_{C_1}\sigma_{C=t}(R)$ is o_1 and the offset to the $t+1$-st fragment is o_2. Then, given a request for the fragment $\pi_{C_1}\sigma_{C=t}(R)$, the lookup table returns the start pointer $\mathcal{F} = s + o_1$ and the size $l = o_2 - o_1$.

Retrieve a fragment $\pi_A\sigma_{F_i=c}(R)$**.** GQ-Fast first obtains an offset-arrays $\mathcal{P} = \mathcal{I}_{R.F_i}[c]$ by a random access, and its neighbor $\mathcal{P}_{next} = \mathcal{I}_{R.F_i}[c+1]$. Then GQ-Fast gets the start address of the fragment $\mathcal{F}_{R.A} = \mathcal{P}[A]$ and its length l=$P_{next}[A] - \mathcal{F}_{R.A}$. If this fragment is encoded by an encoding method **E**, GQ-Fast decodes it by using a macro $decode\mathbf{E}(\mathcal{F}_{R.A}, l : \mathcal{A}_{R.A}, n)$ to produce a decoded array $\mathcal{A}_{R.A}$ and the number of elements n within it.

In the following, we present various encodings for fragments utilized in this paper.

2.4.2 Fragments Encoding Methods

In a GQ-Fast index $\mathcal{I}_{R.C}$ of relation $R(C, C_1, C_2, ..., C_n)$, all the values associated with $t \in C$ in column C_i are organized as a fragment $\pi_{C_i}\sigma_{C=t}R$. GQ-Fast compresses fragments with different compression methods. GQ-Fast does not have any restrictions on compression methods, as long as fragments can be decompressed without accessing other fragments. It allows a wide range of encoding methods, including those that do not support random access within a fragment. GQ-Fast currently uses the following four methods for encoding single fragments. The extended version provides more details on describing the encoding methods.

- *Uncompressed Array (UA):* An uncompressed array stores the original numerical values in their declared type.

- *Bit-aligned Compressed Array (BCA):* Assume a foreign key attribute points to the IDs of an entity, which range from 0 to $h-1$. Then each foreign key value needs $\lceil \log_2 h \rceil$ bits. Consequently, a fragment $\pi_A\sigma_{F=c}R$ with size n requires $\lceil \frac{n \cdot \lceil \log_2 h \rceil}{8} \rceil$ bytes (including

Table 2.1. Space analysis of encoding methods

Uncompressed Array (UA)	$32 \cdot N \cdot \lceil \log_{2^{32}} D \rceil$
Bit-aligned Compressed Array (BCA)	$8 \cdot \lceil \frac{N \cdot \lceil \log_2 D \rceil}{8} \rceil$
Byte-aligned Compressed Bitmap (BB)	$N \cdot (8 \cdot \lceil \log_{128} \frac{D-N}{N} \rceil)$
Huffman	$8 \cdot \lceil \frac{N \cdot E_D + D}{8} \rceil$

alignment-induced padding).

- *Byte-aligned Compressed Bitmap (BB):* Given an array of values $[v_1, \ldots, v_n]$, the equivalent uncompressed bit vector is a sequence of bits, such that the bits at the positions $v_1, \ldots, v_n$ are 1 and all other bits are 0. GQ-Fast uses the byte-aligned method to compress bit vectors [8]. The first bit of a byte is a flag that declares whether (i) the next seven bits are part of a number that also uses consequent bytes or (ii) the remaining seven bits actually represent the length number by themselves.

- *Huffman-encoded Array (Huffman):* GQ-Fast employs Huffman encoding with an array-based encoding of the Huffman tree [25, 62] to avoid tree traversals (i.e., random access on the heap). This can speed up decoding due to CPU L1/L2 caching effects.

We compared the performance and storage tradeoff of all encoding methods analytically and experimentally. Table 2.1 summarizes the space needed by each fragment. Assume that each fragment contains N elements, the domain size of the column containing this fragment is D, $E_D = -\sum_{i=1}^{D} p_i \log p_i$ is the entropy of the column, and p_i is the probability of occurrence of element i[7]. In our experiments, GQ-Fast chooses an optimal encoding for each column with minimal space cost by using the formulas in Table 2.1, where N is set to be the average fragment size on each column.

Figure 2.6 shows the most compact way to encode a fragment in key/foreign key columns, as a function of (a) the number N of elements in the fragment (vertical axis), and (b) the size

[7]We report the lower bound of the space needed by Huffman. The space needed by Huffman is bounded by $[8\lceil \frac{N \cdot E_D + D}{8} \rceil, 8\lceil \frac{N \cdot E_D + N + D}{8} \rceil)$ [71].

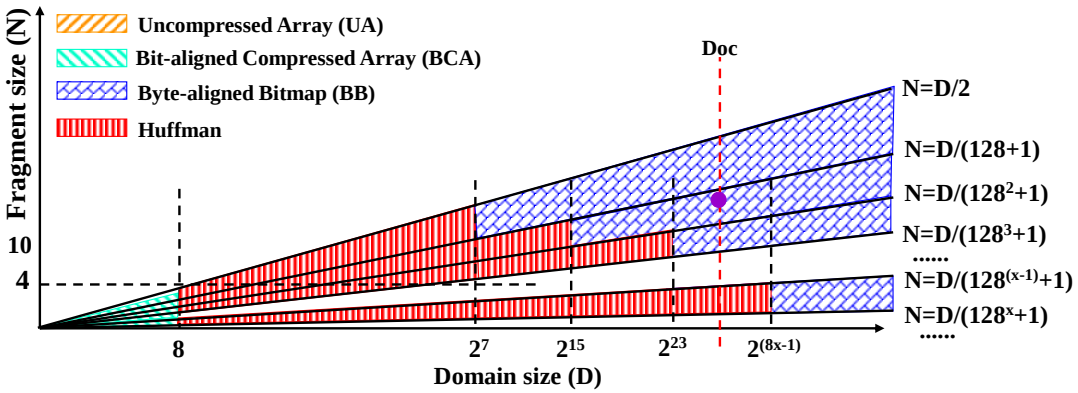


Figure 2.6. Space cost comparison of different encoding methods. Each area is colored by the compression method with minimal space cost.

of the underlying domain D (horizontal axis)[8]. Ideally, each fragment should be encoded by the method with minimal space cost, as indicated by Figure 2.6. However, this leads to space overheads due to recording the encoding method for each fragment. In our experiments, we choose the optimal encoding method for each column based on this figure by setting N as the average fragment size on each column.

Figure 2.6 implies that different fragments of the same column may be most compactly encoded with different methods. For example, a fragment of more documents id's of a term should be encoded with BB, otherwise, it is less suitable to apply BCA. Though applying different encodings for different fragments can achieve minimal space cost, the penalty is that we need to remember the encoding method for each fragment, which increases the space cost. To balance this trade-off, in this paper, we apply the same encoding (the one with minimal space cost for the fragment with average size) for fragments in the same column. Note that, fragments in different columns still benefit from applying different encodings. And only one encoding type is required to store for each column, which can be stored in the metadata.

[8]BCA can only be applied to fragments containing unique values.

2.4.3 Building GQ-Fast Indices

For an entity table $E(ID, M_1, \ldots, M_m)$, GQ-Fast chooses the ID column as the indexed column and creates one index $\mathcal{I}_{E.ID}$. Note that in an entity table, a fragment contains only a single value.

For a relationship table $R(F_1, F_2, M_1, \ldots, M_m)$ with two foreign keys F_1 and F_2, GQ-Fast chooses both F_1 and F_2 as indexed columns, which means GQ-Fast builds two indices $\mathcal{I}_{R.F_1}$ and $\mathcal{I}_{R.F_2}$ according to different indexed columns. The reason is that a relationship table refers to a collection of (potentially undirected) edges in graphs, and it is necessary to provide an efficient way to obtain fragments for both source vertices (in column F_1) and destination vertices (in column F_2). For scenarios where relationship tables have more than two foreign keys, say $a > 2$, to fully index all the foreign key columns (if needed) GQ-Fast builds a indices, which may require a large amount of space.

2.4.4 Incremental Updates

GQ-Fast's compact storage strategy (storing all the fragments of the same attribute in one big fragment array and using offsets to refer to them) can significantly reduce space costs at the expense of incremental updates. To support incremental updates, GQ-Fast could (i) store each fragment independently and (ii) maintain explicit pointers for them. Theoretically, GQ-Fast will then require additional $N(64 - \lceil \log_2 N \rceil)$ bits, where N is the total number of distinct values in the indexed column.

As fragments may be encoded using Huffman encoding, it is challenging to maintain the optimality of Huffman-encoded fragments after massive updates. Dynamic Huffman encoding [54] should be applied, which remains optimal as the weights change.

$$
\begin{array}{lcll}
RQNA & \Rightarrow & \gamma_{k;f_1(.)\mapsto N_1,\ldots,f_n(.)\mapsto N_{n_\gamma}}\, Join & (1)\\
 & & \text{attributes named } k \text{ are primary or foreign keys} & \\
 & | & Join & (2)\\
Join & \Rightarrow & Join \bowtie_{j.k_1=v.k_2} (\pi_{\bar{A}}(T \mapsto v)) & (3)\\
 & & j \text{ is a variable defined by } Join & \\
 & | & \pi_{\bar{A}}(\sigma_c(T \mapsto v)) & (4)\\
 & | & \pi_{\bar{A}}((T \mapsto v) \ltimes_{v.k_1=x.k_2} Context) & (5)\\
 & & x \text{ is a variable defined by } Context & \\
Context & \Rightarrow & \pi_{v.k} Join & (6)\\
 & | & \pi_{v.k}\sigma_{c_1}(T_1 \mapsto v) \cap \ldots \cap \pi_{v.k}\sigma_{c_n}(T_n \mapsto v) & (7)
\end{array}
$$

Figure 2.7. Grammar describing RQNA expressions

2.5 GQ-Fast Query Processing

The GQ-Fast Query Processor (Figure 2.4) transforms a given query into an RQNA expression, which is then transformed into a plan of physical operators (e.g., the plan in Figure 2.9 corresponds to the RQNA expression in Figure 2.8(e)), which is then used together with metadata for C++ code generation. This chapter formally describes RQNA expressions, presents physical operators and key intuitions in the translation of RQNA expressions into plans, and describes how the GQ-Fast code generator translates plans into code, essentially by mapping each physical operator to an efficient code snippet and stitching these snippets together.

2.5.1 RQNA Expression

To efficiently answer relationship queries, GQ-Fast first translates them into RQNA (Relationship Query Normalized Algebra) expressions (Figure 2.7). In the simplest case, an RQNA expression is a left-deep series of joins with a selection and aggregation: In Line 4 the RQNA expression starts with a selection $\sigma_c(T \mapsto v)$ of qualifying entities – we call them the *context* entities[9]. Subsequently, the RQNA expression performs a series of left-deep joins (Line 3) that navigate to entities related to the qualifying entities. Optionally, an RQNA expression may

[9]The condition may be set to true, setting the context to all entities.

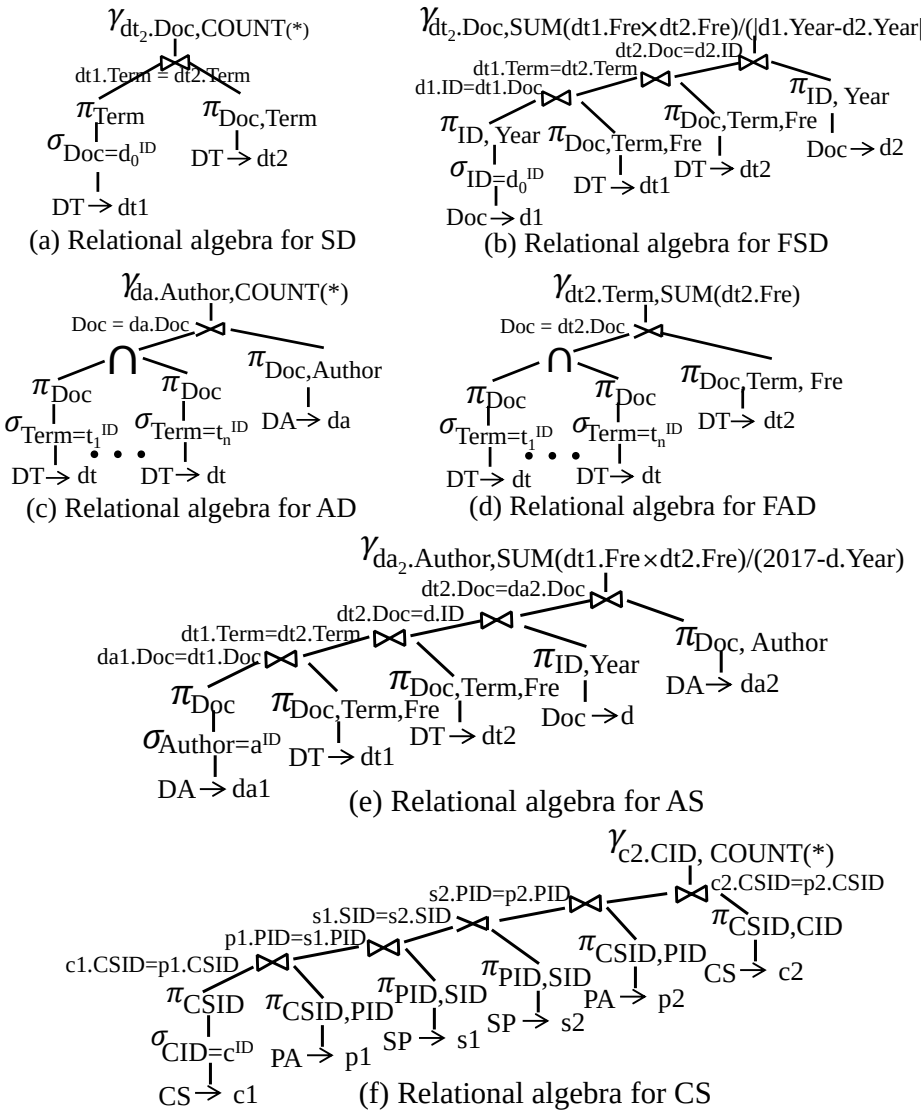


Figure 2.8. Relational algebra expressions for queries on PubMed/SemMedDB

group-by the key attribute k (Line 1), followed by multiple aggregations.

In more complex cases, an SQL query (as shown in later examples) may contain nested queries using **`IN`** syntax, where **`IN`** translates to semijoins (Line 5). Nested queries are themselves relationship queries (Lines 6) without aggregation or the result of an intersection (Lines 7). Figure 2.8 shows the RQNA expressions for all the queries evaluated in this paper.

2.5.2 Physical Operators

This chapter explains the physical operators' syntax and semantics, neglecting for now the bottom-up pipelined execution aspects.

Fragment-based Join. The operator $\overrightarrow{\bowtie}^{r.A_1,\ldots,r.A_n}_{B;R\mapsto r}\ \mathcal{I}_{R.B'}L$ receives as input the result of an expression L that produces a column B, generally among others. For each value $b \in B$, the operator uses the index $\mathcal{I}_{R.B'}$ to retrieve (and decompress) the fragments $\pi_{A_i}\sigma_{r.B'=b}\ (R \mapsto r)$ for $i = 1, \ldots, n$. Intuitively, L would be the left operand of a conventional join and $R \mapsto r$ would be the right side. Conceptually, one may think that the fragments are combined into a result table whose schema has the attributes $A_1, \ldots, A_n$ (and also the attributes of L). However, in reality, the decompressed fragments are not combined into rows. In adherence to the late binding technique [1, 2] of column-oriented processing, the ordering of the items in the fragments dictates how they can be combined into tuples. The $\overrightarrow{\bowtie}$ operator is useful for executing both selections and joins of the RQNA expressions:

- A projection/join combination $\pi_{attrs(L),r.A_1,\ldots,r.A_n}(L \bowtie_{B=r.B'} R \mapsto r)$ where B is an attribute of L and B' is a foreign key of a relationship table R or B' is the ID of an entity table R, translates to $L\ \overrightarrow{\bowtie}^{r.A_1,\ldots,r.A_n}_{B;R\mapsto r}\ \mathcal{I}_{R.B'}$.

- A projection/selection combination $\pi_{r.A_1,\ldots,r.A_n}\sigma_{r.B'=c}(R \mapsto r)$, where c is a constant and B' is a foreign key of a relationship table R or B' is the ID of an entity table R, translates to $\{[B : c]\}\ \overrightarrow{\bowtie}^{r.A_1,\ldots,r.A_n}_{B;R\mapsto r}\ \mathcal{I}_{R.B'}$. Essentially, GQ-Fast reduces the selection into a join, by considering the left-hand-side argument to be a table with a single tuple and a single attribute B, whose value is c.

Fragment-based Semijoin. The operator $\overrightarrow{\ltimes}^{r.A_1,\ldots,r.A_n}_{B;R\mapsto r}\ \mathcal{I}_{R.B'}L$ operates similarly to the fragment-based join but returns only attributes from $(R \mapsto r)$ if there is a matching tuple in L. It is introduced in the plan when the RQNA expression has an expression $\pi_{r.A_1,\ldots,r.A_n}((R \mapsto r) \ltimes_{B=r.B'} L)$. The operator maintains a lookup structure for values from the B column of L; for

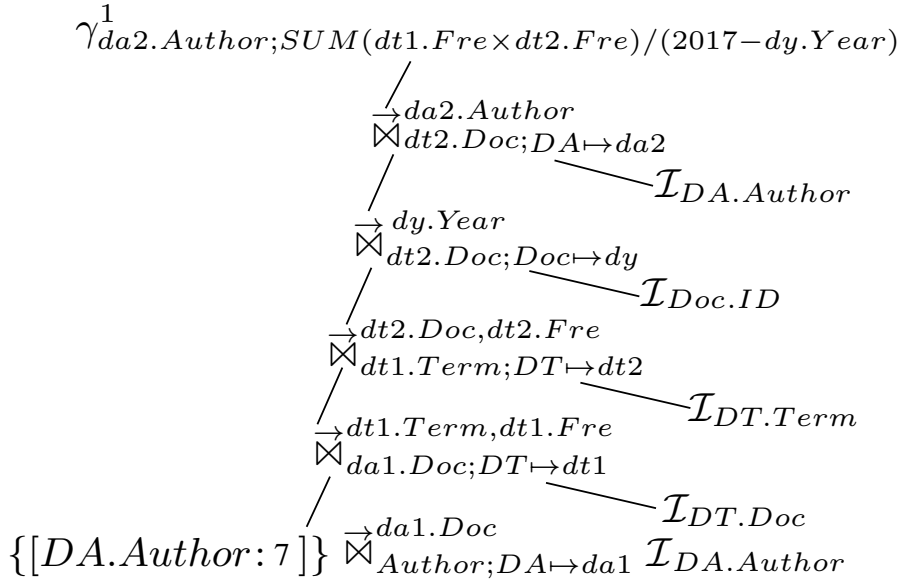


Figure 2.9. Physical algebraic plan for query AS

each value $b \in B$, the operator checks the lookup structure to find out whether that particular value b was already received earlier. If L is relatively large, it is best to use a boolean array, despite the fact that the query needs to initialize all array elements to *false*. Otherwise, a hash set or a tree is preferable.

While joins and semijoins are sufficient, the extended version also describes the occasional replacement of semijoins with a *merge intersection* operator that merges sorted one-attribute relations. As an example, Query AD uses that operator.

Merge Intersection. The operator $\overrightarrow{\bigcap}^{\theta}_{L_1,\ldots,L_m}$ computes the result of $L_1 \cap \ldots \cap L_n$, when each argument L_i consists of a single sorted attribute. In the case of RQNA queries, an L_i meets the necessary conditions if it is an expression of the form $\pi_{l_i.B_i}\sigma_{l_i.K=c}(L_i \mapsto l_i)$, where K is a (foreign or primary) key attribute.

The superscript $\theta = 0$ indicates that the merging should happen directly on the encoded fragments. In a common example, if all the fragments are encoded with compressed bitmap then the merge intersection can be directly applied on the compressed bitmaps. Otherwise, $\theta = 1$ means fragments $L_1 \ldots L_n$ are encoded with different encodings. Then each fragment needs to

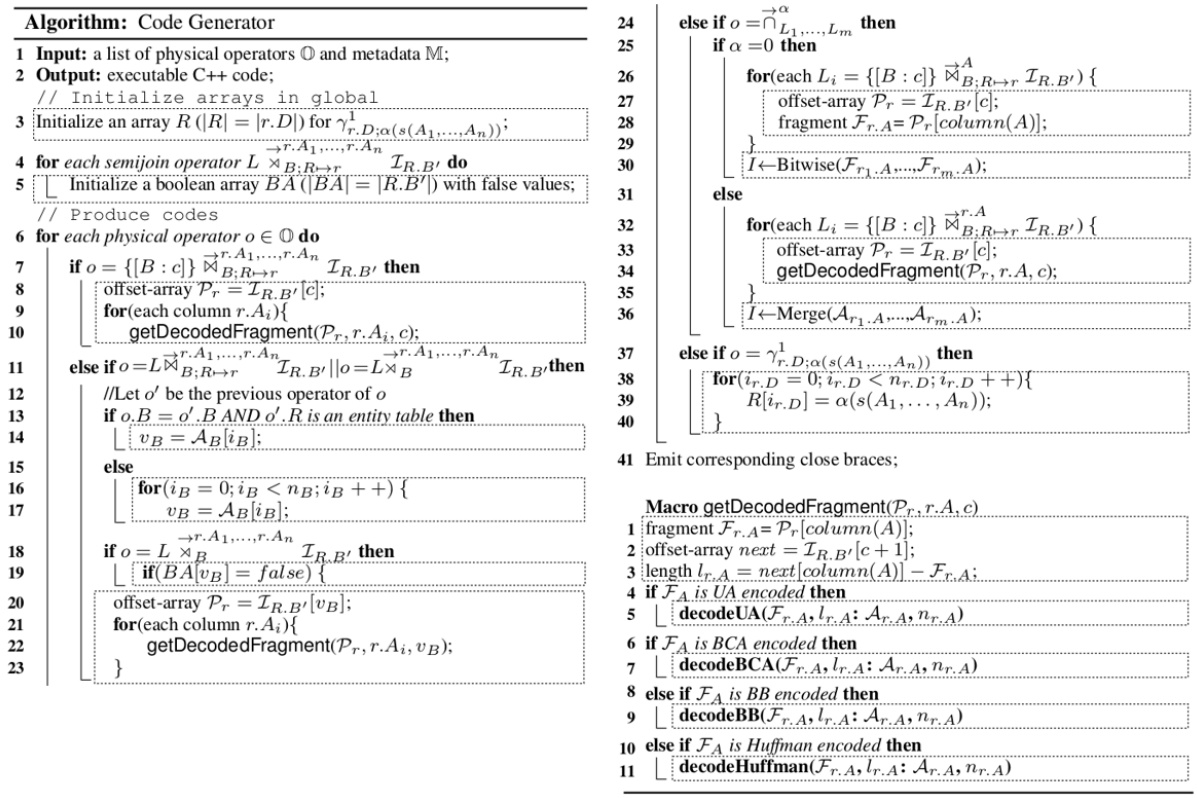
be decoded before intersecting by merging. More precisely, a (preallocated, since we know the worst case scenario in advance) array with size $\min\{|L_1|, \ldots, |L_m|\}$ is initialized to store the final intersection results. The operator has pointers pointing to the beginning of each fragment. The pointed-to values are compared: If all the values are the same, then this value is added to the array and all pointers are incremented. Otherwise, all pointers except the one with maximal value are moved forward until the pointed-to values are greater or equal than the maximal one. Then it repeats the above step until all the values in a fragment have been accessed. The operator $\overset{\rightarrow\theta}{\cap}_{L_1,\ldots,L_m}$ is introduced in the plan when the RQNA expression has a series of semijoins and the right arguments of the semijoins are a single sorted column.

In particular, an RQNA subexpression $\pi_{r.A_1,\ldots,r.A_n}((R \mapsto r) \ltimes_{r.B'=l_1.B_1} \pi_{l_1.B_1} L_1) \ltimes \ldots \ltimes_{r.B'=l_m.B_m} (\pi_{l_m.B_m} L_m)$ translates into $\overset{\rightarrow r.A_1,\ldots,r.A_n}{\rtimes}_{B;R\mapsto r} \mathcal{I}_{R.B'} \overset{\rightarrow\theta}{\cap}_{\pi_{l_1.B_1}L_1),\ldots,\pi_{l_m.B_m}L_m)}$.

Aggregation. The aggregation operator $\gamma^1_{r.D;\alpha(s(A_1,\ldots,A_n))}$ groups its input according to the single group-by attribute $r.D$ and aggregates the results of the scalar function $s(A_1, \ldots, A_n)$ using the associative aggregation function α (e.g., `min`, `max`, `count`, `sum`).

Recall, in relationship queries the single group-by attribute $r.D$ is the foreign key of a relationship table or the ID of an entity. In either case, the range of $r.D$ is the same as the range of the underlying entity ID. Consequently, the γ^1 operator's superscript 1 signifies the assumption that the domain of G is small enough to allow for the allocation of an array, whose size is the domain of $r.D$ and each entry is a number, initialized to zero. Every time a "tuple" from r is processed, this array is updated at $r.D$ accordingly. In addition, an array of booleans registers which values of $r.D$ were actually found. As was also the case with the lookup structure of the semijoin, it is preferable to incur the penalty of initializing such arrays, instead of using hash sets or tree-based sets that have more expensive lookup times. For example, the aggregation $\gamma^1_{Author;\frac{SUM(DT_2.Fre\times DT_1.Fre)}{(2017-DY.Year)}}$ of the Query AS, as shown in Figure 2.9, initializes an array with size of the domain **`Author`** to store the aggregated score for each author and also a boolean register array to check which authors are actually accessed.

In non-relationship queries the group-by list may involve more than one group-by at-



tributes, in which case it uses a hash table [32] instead of an array. In other non-relationship queries, the aggregation function f is non-associative (e.g., the median) in which case it is not enough to allocate an aggregate values' array with just one number per domain entry.

2.5.3 Code Generator

The GQ-Fast code generator has two main components: to-be-emitted source code boxed by dotted lines in the pseudocode; and control commands that determine which code pieces should be emitted. The input of the code generator is (1) the physical plan and (2) GQ-Fast metadata, which specifies the encoding of each fragment. The code generator has two phases: (1) initialize necessary buffers (Lines 3–5); and (2) emit code pieces for each physical operator (Lines 6–40). More precisely, in the first phase, the GQ-Fast code generator initializes an array R to store final aggregation results and several boolean arrays for duplicate-checking in semijoin operations. In the second phase, it emits code for selection operators $\{[B : c]\} \overrightarrow{\bowtie}^{r.A_1,\ldots,r.A_n}_{B;R\mapsto r} \mathcal{I}_{R.B'}$

(Lines 8–10). The getDecodeFragment macro emits code for (1) retrieving a fragment (Lines 1–3) and (2) calling the corresponding decode macro (Lines 4–11); encoding information is obtained from metadata. For each join $L \overset{\rightarrow r.A_1,\ldots,r.A_n}{\bowtie_{B;R\mapsto r}} \mathcal{I}_{R.B'}$ and semijoin $L \overset{\rightarrow r.A_1,\ldots,r.A_n}{\ltimes_{B;R\mapsto r}} \mathcal{I}_{R.B'}$ operator (Lines 11–23), the generator first checks whether the previous operator operates on an entity table and the operated columns are the same. In that case a for-loop can be avoided (Line 14). For a semijoin operator, one more duplicate-checking step should be added (Line 19). The remaining steps (Lines 20-23) of join/semijoin are identical to the selection operator. For each intersection operator $\overset{\rightarrow\alpha}{\cap}_{F_1,\ldots,F_m}$, the generator first identifies whether all the fragments are encoded with the same bitmap encoding (metadata). If so ($\alpha = 0$), the generator emits code to perform intersection directly on encoded fragments (Lines 26–30). Otherwise, it emits code to perform intersection on decoded fragments (Lines 32–36). Then, the code generator emits aggregation code pieces (Lines 38–40) for an aggregation operator $\gamma^{1\alpha}_{r.D;\alpha(s(A_1,\ldots,A_n))}$.

Memory Requirements. Query execution requires $4 \cdot |r.D| + \sum_{i=1}^{k} \cdot |r_i.B'|$ bytes, where $|r.D|$ is the domain size of $r.D$ for an aggregation operator, k is the number of semijoin operators, and $|r_i.B'|$ is the domain size of $r_i.B'$ for the i^{th} semijoin operator.

Parallel Computing. GQ-Fast can use multiple cores/threads to perform parallel computation. The fact that in GQ-Fast (1) each fragment is independent of others, so GQ-Fast can assign fragments to different threads; and (2) the query processing is more CPU-bounded than memory-bounded especially when decompressing encoded fragments. We would like to mention one important technical detail of how to handle two kinds of global arrays, i.e., boolean arrays for each semijoin operation, and a numerical array for aggregation operator. To guarantee the correctness of GQ-Fast, it applies spinlock [13] in each array slot, which is experimentally verified to be more efficient than just using one spinlock on the entire boolean array.

Table 2.2. Data characteristics of PubMed-M and PubMed-MS

Table name	# rows
DT(Doc,Term,Fre)	207,092,075
DT(Doc,Term,Fre)	901,388,401
DA(Doc,Author)	61,329,130
Document(ID,Year)	23,176,635

Entity ID	Domain Size
Doc(ument)	23,326,299
Term	27,883
Term	259,728
Author	6,301,521

Table	Fragment	Average size	Maximal size	Standard deviation
DT	**Doc**	7427.18	8192342	197.56
	Term	14.48	667	17.52
DT	**Doc**	3470.50	8192342	318.72
	Term	63.06	753	39.06
DA	**Doc**	5.99	5712	21.93
	Author	4.35	3163	8.04
Document	**Year**	1.00	1	0.00

2.6 Experiments

We evaluated GQ-Fast's novelties by running relationship queries on three real-life datasets. In all experiments, the entire data set was located in main memory.

2.6.1 Environment and Setting

All experiments were done with GQ-Fast 0.1 on a computer with a 4th generation Intel i7-4770 processor (4 $\times$ 32 KB L1 data cache, 4 $\times$ 256 KB L2 cache, 8 MB shared L3 cache, 4 physical cores, 3.6 GHz), 16 GB RAM, and a Seagate ST2000DM001-1CH1 hard drive, running Ubuntu 14.04.1. Generated C++ code was compiled with g++ 4.8.4, using `-O3` optimization.

Dataset. We evaluated all selected DB systems and design choices with three datasets: PubMed-M, PubMed-MS and SemMedDB. Table 2.2 [10] and Table 2.3 summarize their data characteristics (their schemas are presented in Chapter 2.1 and Chapter 2.2).

Compared Systems. To provide an end-to-end comparison, we compared GQ-Fast with

[10]PubMed-MS has more terms than PubMed-M, which results in larger size of **DT** table in PubMed-MS. Gray cells indicate the difference between them.

Table 2.3. Data characteristics of SemMedDB

Table	# rows	Table	# rows
CS	1550482	**Concept**	1339227
PA	37508726	**Sentence**	146055876
SP	81929321	**Predication**	17359895

Table	Fragment	Ave size	Max size	Standard deviation
CS	**concept_semtype_id**	1.16	5.00	0.39
	concept_id	1.00	1.00	0.00
PA	**predication_id**	122.00	109532	845.15
	concept_semtype_id	2.15	38	0.53
SP	**sentence_id**	4.65	125367	112.36
	predication_id	1.61	140	1.07

the graph database *Neo4j* 2.3.2 [11], the row-oriented database *PostgreSQL* 9.4.0, the cluster-based and column-oriented *Vertica* Analytics Platform, and the in-memory column database *MonetDB*. We measure the *warm* running time for queries, i.e., each query is run twice and we only report the second measurement. The first one is used just to bring all the necessary data (for the evaluation of the query) into the RAM buffers. Furthermore, we run and average five warm runs.

To isolate the effect of compiled code from the other contributions of GQ-Fast, we implemented two main-memory column databases serving as main-memory baselines. One is a *plain main-memory column database (PMC)* without optimizations. The other one is a *fully optimized main-memory column database (OMC)*. They both utilize a code generator for executable C++ plans. The logical query plans in PMC and OMC are identical to the ones in GQ-Fast (same RQNA expressions). Both PMC and OMC use the operator-at-a-time execution model as MonetDB [94, 3] does. PMC maintains one copy of each unsorted table, and uses whole column scans when executing each operator. OMC maintains two copies of each table, such that each copy is sorted based on one foreign key column. OMC applies all optimizations that can improve the performance of relationship queries: (1) Applying run-length encoding for

[11] Since Neo4j does not support SQL syntax, we translated queries into Cypher, Neo4j's query language.

sorted columns, improving the lookup performance and reducing space costs; (2) utilizing binary search for sorted columns instead of whole column scan.

2.6.2 Experimental Results

We ran the Queries SD, FSD, AD, FAD and AS on PubMed (PubMed-M and PubMed-MS) and Query CS on SemMedDB (see queries in Chapter 2.2). We always chose the encoding with the least space costs, if not stated otherwise; even though a different encoding might perform better in terms of running time. We measured the *warm* running time for queries, i.e., each query was run twice but only measured the second time.

We measured the overall runtime performance (Chapter 2.6.2.1) and the overall space cost (Chapter 2.6.2.2) for each algorithm and database. The results show that GQ-Fast outperforms MonetDB and OMC by 10–10^3 and 7–70 times, respectively, and generally uses less space, due to a combination of the following effects:

- *Compilation*: Using a code generator to generate C++ code.
- *Pipelining*: Adopting a bottom-up pipelined execution strategy.
- *Array-l*: Using dense IDs to maintain an array look-up table instead of a hash table.
- *Array-a*: Using dense IDs to maintain an array to store aggregation results instead of hash table.
- *Compression*: Applying aggressive data compression schemes.

The gap between the speedup of GQ-Fast and OMC over MonetDB reveals the power of compiled code. In order to isolate the effect of the other four optimizations, we implemented variants of GQ-Fast and OMC as summarized in Table 2.4.

GQ-Fast-UA is GQ-Fast with uncompressed arrays (as encoding). In addition, GQ-Fast-UA(Bin) uses binary search instead of array lookup. Therefore, GQ-Fast-UA(Bin) does not have the dense IDs optimization. GQ-Fast-UA(Map) is like GQ-Fast-UA but uses a hash map

Table 2.4. Summary of different variants of GQ-Fast and OMC

	Compile	Pipeline	Array-l	Array-a	Compress
GQ-Fast	✓	✓	✓	✓	✓
GQ-Fast-UA	✓	✓	✓	✓	✗
GQ-Fast-UA(Bin)	✓	✓	✗	✓	✗
GQ-Fast-UA(Map)	✓	✓	✓	✗	✗
OMC	✓	✗	✗	✗	✓[12]
OMC-denseID	✓	✗	✓	✓	✓[13]

instead of an array to store final aggregation results. It does not use the dense IDs optimization. OMC-denseID is like OMC but uses arrays instead of hash maps in both lookup and aggregation, which means OMC-denseID has the same lookup and aggregation data structures as GQ-Fast.

To isolate the effect of each single optimization, we conduced further experiments as described in Chapter 2.6.2.3.

- To measure the effect of dense IDs, we compared (i) GQ-Fast-UA with GQ-Fast-UA(Bin) (Table 2.8), and (ii) GQ-Fast-UA with GQ-Fast-UA(Map) (Table 2.9).
- To measure the effect of using bottom-up pipelining against materializing intermediate results, we compared GQ-Fast-UA with OMC-denseID (Table 2.10).
- To measure the effect of applying different compressions, we analyzed the performance and space cost of different compressions in GQ-Fast (Table 2.11).

Chapter 2.6.2.4 provides additional experiments to analyze (1) the effect of parallel processing in GQ-Fast, and (2) the time required for building GQ-Fast indices.

2.6.2.1 Overall Runtime Performance

Table 2.6 reports the average running time of each query for each system, using 8 threads[14]. Overall, GQ-Fast shows superior performance for all queries. We further observed

[12]OMC uses RLE encoding and dictionary encoding.

[13]OMC-denseID uses RLE encoding and dictionary encoding.

[14]We applied spinlocks [13] to ensure correctness during concurrent access to shared arrays for the semijoin and aggregate operators.

Table 2.5. Running time on general graphs

	SD	AD	AS
GQ-Fast on general-graphs	1.440	0.407	47.077
Neo4j on general-graphs	137.6	80.2	50121.9

that:

- On average, GQ-Fast outperforms Vertica, MonetDB and OMC by a factor of 100, 170, and 20, respectively (see ratio columns). If GQ-Fast only applies UA compression, it will achieve better performance (running time of Query AS on PubMed-M is 4.45s).
- MonetDB outperforms PMC: Its indexed plans perform better than PMC's compiled code. OMC outperforms MonetDB, since (a) OMC uses code generation and (b) has two copies of each relationship table. For example, OMC uses two copies of the **`DT`** table in Query SD. Therefore, each OMC lookup is a binary search on the sorted column (hence essentially tieing the index-based lookups of MonetDB) and lookup results are run-length encoded on the sorted column, hence reducing the size of intermediate results.
- High fanout is favorable to GQ-Fast: The improvement over the competing systems is usually higher in the queries SD, FSD, FAD and AS when they use **`DT`** of PubMed-M, compared to queries that use **`DT`** of PubMed-MS. **`Term`** has a higher fanout in **`DT`** of PubMed-M. We conjecture that high fanouts ammortize over larger fragments the fixed costs of the decompression routines, therefore extending GQ-Fast's advantages.

We also conducted experiments to evaluate the performance of GQ-Fast on general graphs. As shown in Table 2.5, GQ-Fast is still about 100x faster than Neo4j, even after incurring the 10x slowdown (due to lack of knowledge on types), which speaks to the applicability of the GQ-Fast techniques in the case of general graphs.

Table 2.6. End-to-end runtime performance (in seconds). Numbers in bold are the fastest ones.

	Query	(a) Neo4j	(b) Postgres	(c) Vertica	(d) MonetDB	(e) PMC	(f) OMC	(g) GQ-Fast	MonetDB / GQ-Fast	OMC / GQ-Fast
PubMed-M	SD	14.7	211.2	5.19	10.8	23.36	2.47	**0.230**	47.0	10.7
	FSD	86.6	567.5	12.32	23.9	33.98	5.93	**0.821**	29.1	7.2
	AD	7.4	158.1	2.17	6.2	4.82	0.73	**0.037**	86.5	19.7
	FAD	18.2	198.6	3.85	8.5	7.16	0.88	**0.064**	70.3	13.8
	AS	5546.5	29520.5	42.34	4474.8	5832.30	194.77	**5.662**	790.3	34.4
PubMed-MS	SD	61.6	741.2	37.17	49.3	400.24	10.25	**1.068**	46.2	9.6
	FSD	146.8	2148.7	53.48	112.8	1892.30	32.80	**4.376**	25.8	7.5
	AD	6.9	112.9	5.60	5.2	19.67	0.67	**0.035**	77.1	19.1
	FAD	11.1	119.6	15.17	3.5	24.99	0.71	**0.062**	56.5	11.5
	AS	9604.8	180164.1	362.35	28918.8	33321.74	3083.30	**54.720**	528.5	56.3
SemMedDB	CS	21.0	53.1	10.94	4.7	23.58	2.12	**0.031**	151.6	68.4

Table 2.7. Space cost for each system (in GB). Numbers in bold are the smallest ones

	(a) Neo4j	(b) Postgres	(c) Vertica	(d) MonetDB	(e) PMC	(f) OMC	(g) GQ-Fast	MonetDB / GQ-Fast	OMC / GQ-Fast
PubMed-M	34.36	20.92	3.21	3.69	3.09	3.49	**1.47**	2.51	2.37
PubMed-MS	112.15	78.90	11.06	13.27	11.42	11.82	**3.51**	3.78	3.37
SemMedDB	10.39	6.84	2.89	1.23	**0.97**	2.05	1.36	0.90	1.51

2.6.2.2 Overall Space Cost

Table 2.7 presents the overall space costs. GQ-Fast has the lowest space cost in PubMed-M and PubMed-MS. Interestingly, GQ-Fast also uses much less space than PMC even though PMC stores only one copy of each table while GQ-Fast stores two "copies" (i.e., two indices); this indicates the importance of dense compressions. In SemMedDB, GQ-Fast still uses less space than OMC, but more space than PMC. The reason is the fanout of SemMedDB (averaging at 1.16), which dilutes the effect of fragment compression since fragments are very small and space is spent on padding them to full bytes. Even though PMC uses marginally less space than GQ-Fast in SemMedDB, GQ-Fast is still the best overall choice as it is 760 (i.e., 23.58/0.031) times faster (Table 2.6).

Table 2.8. GQ-Fast-UA vs. GQ-Fast-UA(Bin) (in ms). The last column shows the improvement, where $\theta = 1 - \frac{\text{GQ-Fast-UA}}{\text{GQ-Fast-UA(Bin)}}$

	Ave # lookups	GQ-Fast-UA(Bin)	GQ-Fast-UA	θ
SD	22	247.94	**177.08**	28.58%
FSD	21748262	1129.72	**435.60**	61.44%
AD	23609	38.67	**30.33**	21.57%
FAD	23609	27.84	**25.95**	6.79%
AS	58589421	7364.92	**4510.11**	38.76%
CS	132975	16.21	**8.62**	46.82%

2.6.2.3 Effect of Each Optimization

Effect of Dense IDs. The dense IDs assumption allows GQ-Fast to use arrays for semijoins and aggregations instead of other data structures like hash maps.

GQ-Fast-UA vs. GQ-Fast-UA(Bin). We conducted experiments to evaluate the performance of retrieving fragments. Table 2.8 shows the running time of different queries for GQ-Fast-UA(Bin) and GQ-Fast-UA on PubMed-M and SemMedDB[15]. GQ-Fast-UA outperforms GQ-Fast-UA(Bin) for all queries. For example, GQ-Fast-UA saves around 12% running time over GQ-Fast-UA(Bin) for Query AS. In addition, we also observed that, queries with larger number of lookup requests (FSD, AS and CS) benefit more compared to queries with smaller number of lookup requests, e.g., SD and AD.

GQ-Fast-UA vs. GQ-Fast-UA(Map). We measured the benefit of choosing an array for aggregation in GQ-Fast over a hash map by comparing GQ-Fast-UA with GQ-Fast-UA(Map). As shown in Table 2.9, GQ-Fast-UA outperforms GQ-Fast-UA(Map) for all queries. GQ-Fast-UA performs better for the queries with a large output (e.g., Query AS; GQ-Fast-UA saves about 33% of running time) compared to queries with smaller output (e.g., Query CS).

Effect of Pipelining. In this experiment we compared OMC-denseID with GQ-Fast-UA in order to measure the benefit of pipelining over materializing intermediate results. OMC-denseID and GQ-Fast-UA have the same lookup data structure and use an array for final

[15]We achieve similar improvements in PubMed-MS.

Table 2.9. GQ-Fast-UA vs. GQ-Fast-UA(Map) (in ms). The last column shows the improvement where $\theta = 1 - \frac{\text{GQ-Fast-UA}}{\text{GQ-Fast-UA(Map)}}$

	Ave # results	GQ-Fast-UA(Map)	GQ-Fast-UA	θ
SD	27,443,100	908.95	**177.08**	80.52%
FSD	27,307,529	1342.82	**435.60**	67.56%
AD	200,679	34.84	**30.33**	12.94%
FAD	56,518	31.63	**25.95**	17.96%
AS	20,019,297	7766.83	**4510.11**	41.93%
CS	5,057	10.06	**8.62**	14.31%

Table 2.10. Avoiding intermediate results

	# fragments	# elements	OMC-denseID	GQ-Fast-UA
A_1	7,484,532	51,730,682	4.12	1.02
A_2	9,287,804	65,687,183	22.15	2.24
A_3	87,467,470	619,809,092	74.90	5.38
A_4	184,219,134	1,305,764,797	171.33	13.76
A_5	585,932,678	4,153,322,719	297.47	49.01

aggregation. Table 2.10 reports the running time of GQ-Fast-UA and OMC-denseID on five instances of Query AS, where each instance queries for another author ID, A_1–A_5. The number of accessed fragments for those queries varies from around 7M to 585M. As shown, GQ-Fast-UA outperforms OMC-denseID by a factor of 15. As the number of accessed elements increases, the running time of OMC-denseID increases significantly, since OMC-denseID materializes larger intermediate result columns.

Analysis of Different Encoding Methods. We analyzed the performance (compression rate and decompression time) of all encoding methods that are employed by GQ-Fast: uncompressed array (UA), bit-aligned compressed array (BCA), byte-aligned bitmap (BB) and Huffman encoding. Table 2.11 reports the encoded size of each column in the PubMed-MS dataset. As shown, no single encoding is optimal for all columns. Adopting a suitable compression method can significantly save space. For example, by using BB, the space cost of **`dt1.Term`** reduced from 3660.29 MB to 1431.12 MB. The selection of a suitable compression method is based on

Table 2.11. Size of encoded columns (MB). The bold fonts show the minimal space for each column. BB only applies for fragments with unique values, so **`dt1.Fre`** and **`dt2.Fre`** are not encoded by BB.

	UA	BCA	BB	Huffman
`dt1.Term`	3605.55	2033.25	**1376.39**	1565.60
`dt1.Fre`	901.39	454.12	N/A	**142.46**
`dt2.Doc`	3605.55	2816.93	**1047.71**	2779.37
`dt2.Fre`	901.39	450.74	N/A	**134.84**
`da1.Doc`	245.26	198.75	**187.54**	325.70
`da2.Author`	245.26	**183.95**	205.10	275.56
`dy.Year`	57.17	**14.20**	N/A	14.29

Table 2.12. Space cost and decompression time for BCA, BB, and Huffman. Domain size is 1 billion, data follows Zipf distribution with factor $s = 1.5$. Fragments only contain unique values, which simulates fragments in foreign-key columns.

	# elements per fragment	# fragments	compression ratio	1 thread	2 threads	4 threads	8 threads
BCA	$100000_{\pm 1000}$	8000	76.23%	1535.51	864.60	450.89	378.23
BB	$100000_{\pm 1000}$	8000	31.75%	1501.81	835.15	428.82	371.44
Huffman	$100000_{\pm 1000}$	8000	73.08%	52198.71	29688.66	14934.78	7925.45

our analysis results in Chapter 2.4. For example, as we discussed, UA always uses most space. In addition, BB achieves the minimal space on **`dt2.Doc`**, which indicates the correctness of our analysis.

Based on the analysis results, the best encodings for **`DT.Term`**, **`DT.Doc`** and **`DA.Doc`** are byte-aligned compressed bitmaps (BB), while the best one for **`DA.Author`** is bit-aligned dictionary (BD). For example, the domain size of **`DT.Term`** is $223705 < 2^{23}$. As shown in Table 2.2, the average size of fragment is 22.51. Therefore, for **`DT.Term`**, BB should be the best one theoretically.

We also conducted experiments to evaluate the decompression performance of these encoding methods for two kinds of (synthetic) fragments: fragments on foreign key columns containing only unique values (Table 2.12) and fragments on measure attributes with many duplicates (Table 2.13). In the former case, we observed that BB achieves the highest compression

Table 2.13. Space cost and decompression time for BCA and Huffman. Domain size is 100, data follows Zipf distribution with factor $s = 1.5$. Fragments contain duplicates, which simulates fragments in measure attributes.

	# elements per fragment	# fragments	compression ratio	1 thread	2 threads	4 threads	8 threads
BCA	100	8000000	21.88%	1581.17	801.31	410.04	348.42
	10000000	80	21.88%	1286.58	652.02	333.65	283.51
Huffman	100	8000000	12.28%	5055.16	2543.28	1280.83	668.05
	10000000	80	11.39%	4374.84	2201.00	1108.45	578.14

Table 2.14. Running time for building indices (seconds)

	(1) GQ-Fast-UA	(2) GQ-Fast	$\frac{(1)+r}{(2)+r}$
PubMed-M	152.52	153.89	73.74%
PubMed-MS	540.33	561.02	72.54%
SemMedDB	74.68	103.30	66.15%

(saving 69.25% space) and the highest decompression performance (about 30 times faster than Huffman). Huffman has the worst performance, since the domain size is large, which requires maintaining a large decoding table (tree) that is too big for CPU L1/L2 caches. In the latter case, we noticed that Huffman achieves the highest compression quality and has decompression performance comparable to BCA, as shown in Table 2.13. Compared to the results in Table 2.12, the decompression performance of Huffman improved significantly, because the Huffman table can fit into the L1 cache when the domain size is small (say 100). This result also indicates that Huffman is suitable for measure attributes.

2.6.2.4 Addition Experiments

Effect of Multiple Threads. We evaluated the effect of multiple threads for the overall performance. Figure 2.10 shows the running time of selected queries with 1–8 threads. Parallel processing improves performance but does not scale linearly, mostly due to skewed data. The skewed problem can be solved by employing load-balance algorithms.

Building Indices. Table 2.14 reports the time for building indices in-memory. For GQ-Fast, this process takes a bit more time than for GQ-Fast-UA, since it spends extra time on

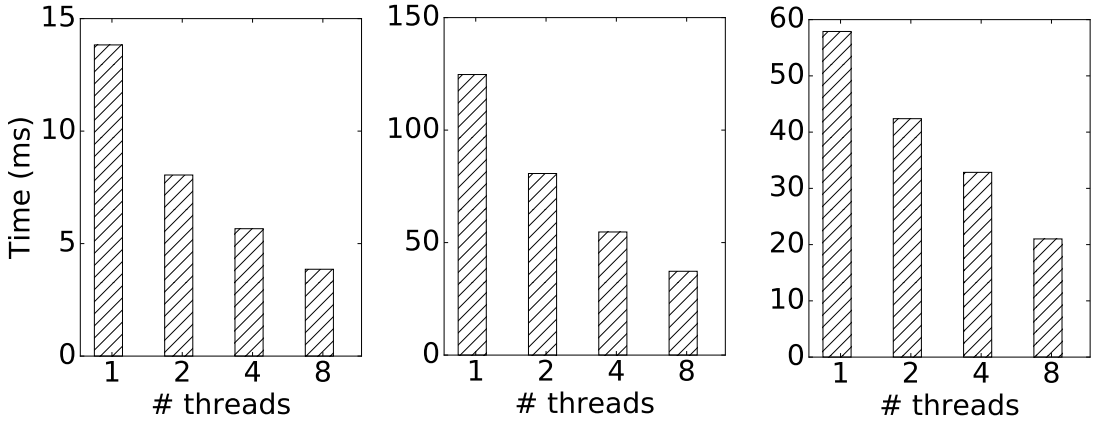


Figure 2.10. Running time for query AS (PubMed-M, PubMed-MS and Query CS (SemMedDB), 1–8 threads

encoding fragments. The last column shows the ratio, where r is the time required for reading data from disk to memory, which is 361.14s for PubMed-M, 1283.32s for PubMed-MS, and 149.77s for SemMedDB.

2.7 Related Work

GQ-Fast Indices vs. Database Indices. Database indices (B+ trees, hash indices and bitmaps) are built on top of the original tables, i.e., the database has both tables and indices. In contrast, the entire dataset of a GQ-Fast database is in GQ-Fast indices. This difference has repercussions in query processing. A database index is given a key and returns row IDs. Depending on the system, a row ID may be a tuple pointer, a block pointer or an array index. At any rate, the database then accesses the tuples that are identified by the row IDs and collects the relevant attributes in the original tables. In contrast, the GQ-Fast index is a data-to-data index, which gets rid of row IDs. Given a key and an attribute, it returns directly the attribute values that relate to the particular key.

Comparison of Pipelining Methods. In row-oriented databases, the top-down iterator model [33] reduces the memory footprint of intermediate results. However, the top-down iterator model shows poor performance on modern CPUs due to lack of locality, frequent instruction

mispredictions and too many function calls [73]. Therefore, modern column databases choose either to (1) pass blocks of tuples (batch-oriented processing) between operators, reducing the number of function invocations [75, 73], or (2) materialize all intermediate results to eliminate the need of calling an input operator repeatedly, which simplifies operator interaction [66, 40], or (3) choose a middle way by passing large vectors of data and evaluating queries in a vectorized manner on each chunk [106].

However, none of the above techniques reaches the speed of hand-written code [73]. GQ-Fast's code generator compiles physical plans to code to improve performance. While many aspects of GQ-Fast code generation (e.g., function call avoidance) have been employed in previous work employing code generation[16]

GQ-Fast produces code that is very close to what a human would do. Crucially, the compiled code utilizes simple for-loops (e.g., see Generated Code in Chapter 2.1) that access the elements in a fragment. Tight for-loops create high instruction locality which eliminates the instruction cache-miss problem. Intermediate results are stored in loop variables, such as the $v_{\texttt{dt1.Term}}$. Such simple loops are amenable to compiler optimizations (e.g., register allocation of loop variables) and CPU out-of-order speculation [40].

Data Cubes vs. Graph Analytics. The fact table of data cubes involves typically $k > 2$ foreign keys. Hence, if we perceive the fact table as a k-ary relationship, we would create k GQ-Fast indices, inducing data redundancy that would eventually surpass the compression advantages. Alternately, we could think of facts as entities, connected to the dimensions via many-to-one relationships. However, once we model a data cube in this way and apply GQ-Fast to it, the benefit of GQ-Fast in performing paths of many-to-many joins is not exhibited anymore. Hence, it becomes apparent that data cube queries and graph queries are significantly different in their SQL OLAP needs and GQ-Fast is tuned towards the latter.

Graph Processing. High-level graph engines allow users to write in SQL or other

[16] Among others, generating code is used to speed up data cube queries [73], view maintenance [7] and path-counting queries [70].

declarative languages, e.g., Datalog, which is easier to use but orders of magnitude slower [4] than low-level graph engines [38, 74]. GQ-Fast meets the performance of low-level graph engines while supporting a high-level programming interface. It is worth mentioning that, EmptyHeaded [4] has the same design goal as GQ-Fast but they have several differences: (1) GQ-Fast uses an encoded fragment-based data structure, while EmptyHeaded employs a trie data structure; and (2) GQ-Fast focuses on CPU caching effects, while EmptyHeaded focuses on leveraging SIMD (single-instruction multiple-data) to speed up performance.

2.8 Chapter Summary

In this chapter, we studied the relationship queries. GQ-Fast used a new fragment-based data structure and a new, coordinated bottom-up pipelining execution strategy to answer relationship queries. Further, it employed a code generator to produce efficient compiled codes. We itemized the sources of GQ-Fast superior performance and measured how much each one of them contributes to the overall performance speedup.

This chapter contains material from "Fast In-Memory SQL Analytics on Typed Graphs" by Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou and Matthias Springer, which appears in Proceedings of VLDB Endowment, Volume 10, Number 3, November 2016. The dissertation author was the primary investigator of this paper.

Chapter 3

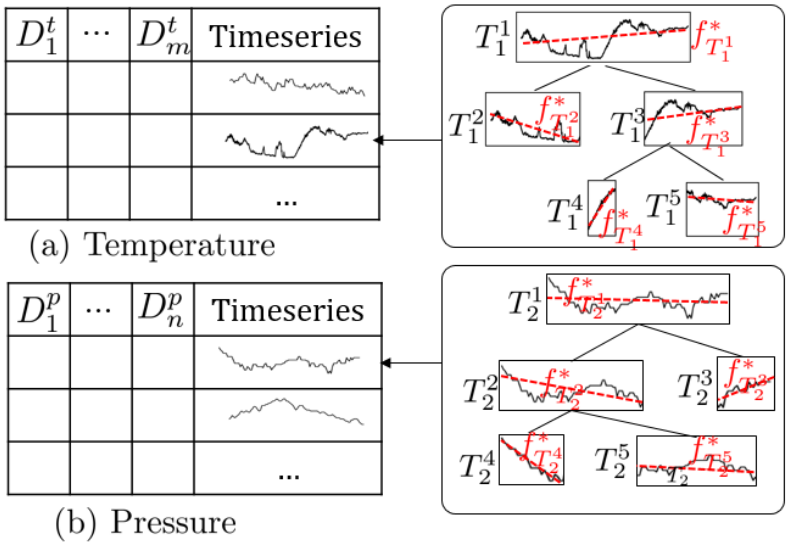
Plato: Supporting Anytime Queries over Compressed Time Series with Deterministic Error Guarantees

3.1 Introduction

Attention to time series analytics is bound to increase in the IoT era as cheap sensors can now deliver vast volumes of many types of measurements. The size of the data is also bound to increase. E.g., an IoT-ready oil drilling rig produces about 8 TB of operational data in one day. [1] One way to solve this problem is to increase the expense in computing and storage in order to catch up. However, in many domains, the data size increase is expected to outpace the increase of computing abilities, thus making this approach unattractive [31, 17]. Another solution is *approximate analytics* over compressed time series.

Approximate analytics enables fast computation over historical time series data. For example, consider the database in Figure 3.1, which has a Temperature table and a Pressure table. Each table contains (i) one Timeseries column containing time series data, as a UDT [28] and (ii) several other "dimension" attributes D, such as geographic locations and other properties of the sensors that delivered the time series. The Plato SQL query in Figure 3.1(c) "returns the top-10 temperature/pressure 5-second cross-correlation scores among all the (temperature,

[1] https://wasabi.com/storage-solutions/internet-of-things/




```
SELECT TSA('(Sum((t.timeseries-Constant(μ(t.timeseries)))×
            (Shift(p.timeseries, 5)-Constant(μ(p.timeseries))))
          )/(σ(t.timeseries)× σ(p.timeseries))',
                t.timeseries, p.timeseries)AS result
FROM Temperature t, Pressure p
WHERE cond(t.D_1^t, ...t.D_m^t, p.D_1^p, ..., p.D_n^p)
ORDER BY approximateAnswer(result)
LIMIT 10;
WITHIN ERROR 0.005;
```

(c) Query

Figure 3.1. Example of SQL query using the TSA UDF.

pressure) pairs satisfying a (not detailed in the example) condition over the dimension attributes". Notice, the first argument of the TSA UDF is a *time series analytic expression* (in red italics). We could write simply 'CCorr(t.timeseries, p.timeseries, 5)', as there is a built-in cross-correlation expression CCorr but, instead, the example writes the equivalent expression that uses more basic functions (such as the average μ, the standard deviation σ and the time Shifting) to exhibit the ability of Plato to process expressions that are compositions of well-known arithmetic operators, vector operators, aggregation and time shifting. Either way, computing the accurate cross-correlations would cost more than 10 minutes. However, Plato reduces the running time to within one second by computing the approximate correlations. It also delivers deterministic error guarantees. Notice that Plato allows users to specify error budgets and returns approximate

answers with deterministic error guarantees no greater than the error budgets. For example, the Plato SQL query in Figure 3.1(c) specifies the error budget to be 0.005. We will show later Plato returns an approximate answer with an error guarantee 0.0032, which is less than the error budget 0.005. (In SQL, the result is a string concatenation of the approximate answer and the error guarantee. The functions approximateAnswer and guarantee extract the respective pieces.)

The success of approximate querying on IoT time series data is based on an important beneficial property of time series data: the points in the sequence of values normally *depend* on the previous points and exhibit *continuity*. For example, a temperature sensor is very unlikely to report a 100 degrees increase within a second. Therefore, in the signal processing and data mining communities [47, 43, 29, 14], time series data is usually modeled and compressed by continuous functions in order to reduce its size. For instance, the Piecewise Aggregate Approximation (PAA) [47] and the Piecewise Linear Representation (PLR) [43] adopt polynomial functions (0-degree in PAA and 1-degree in PLR) to compress the time series; [76] uses Gaussian functions; [96] applies natural logarithmic functions and natural exponential functions to compress time series. Plato is open to any existing time series compression techniques. Notice that there is no one-size-fits-all function family that can best model all kinds of time series data. For example, polynomials and ARMA models are better at modeling data from physical processes such as temperature [68, 24], while Gaussian functions are better for modeling relatively randomized data [52] such as stock prices. How to choose the best function family has been widely studied in prior work [82, 102, 27, 56] and recent efforts even attempt to automate the process [57]. We assume that the Plato users make a proper selection of how to model/compress the time series data and we do not further discuss this issue.

Architecture. Figure 3.2 shows the high-level architecture. During insertion time, the provided time series is compressed. In particular, a compression *function family* (e.g., 2nd-degree polynomials) is chosen by the user. Internally, each time series is partitioned, compressed, and organized as a segment tree structure. Chapter 3.3 presents the details of the segment tree structure and also the segment tree building algorithm. Figure 3.3 gives an example segment

tree structure. Each node in the tree corresponds to a time series segment. For each segment T_i^j Plato finds the best *estimation function* $f^*_{T_i^j}$, which is the member of the function family that best approximates the values in this segment. The most common definition of "best" is the minimization of the *reconstruction error*, i.e., the minimization of the Euclidean distance between the original and the estimated values. This is also the definition that Plato assumes. Plato stores the coefficients of the estimation function for each segment, which take much less space than the original time series data. In addition, Plato also computes *error measures* $\Phi(T_i^j)$ (one to three parameters depending on the selected function families) for each segment.

Consequently, given a query q with TSA UDF calls [2] and an error budget, Plato is able to give an approximate answer with a tight deterministic error guarantee (no greater than the error budget) by accessing the minimal number of nodes in the trees. Plato accesses the segment trees in a top-down way. It recursively picks a node to access its children nodes and computes the current error guarantee (Chapter 3.4 shows how to compute error guarantees by using error measures stored in the current nodes.) until the current error guarantee is no greater than the given error budget. Chapter 3.5 introduces the detailed top-down segment tree navigation algorithm. Note, the TSAs may combine multiple time series; e.g., a correlation or a cross-correlation.

Example 1. *Consider a room temperature time series T_1 and an air pressure time series T_2 in Figure 3.4 and consider the* TSA*('Ccorr(T_1, T_2, 60)', T_1, T_2) where 'Ccorr($T_1, T_2, 60$)' refers to the 60-seconds cross-correlation of T_1 and T_2 (see definition in Table 3.5) along with an error budget* 0.005*. Both T_1 and T_2 have* 600 *data points at 1-second resolution and are segmented by variable length segmentation methods and compressed by PLR (1-degree polynomial functions). The precise answer is* 0.303 *which can be obtained by accessing the* 1200 *(600×2) original data points. Instead of accessing the* 1200 *original data points, Plato accesses the segment trees in a top-down way. Assume Plato now is accessing the nodes with grey background*

[2] We focus on aggregation queries whose results are single scalar values, so the approximate answers are also scalar values.

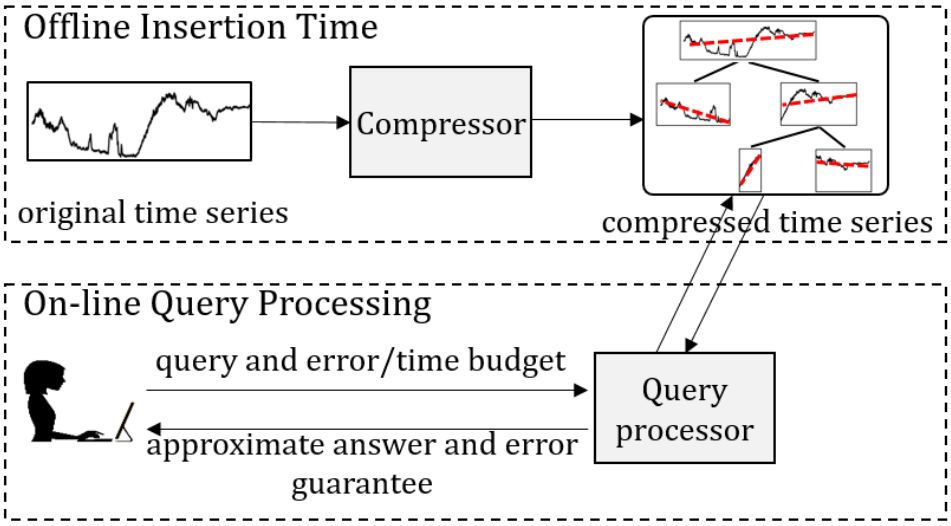


Figure 3.2. Plato's approximate querying

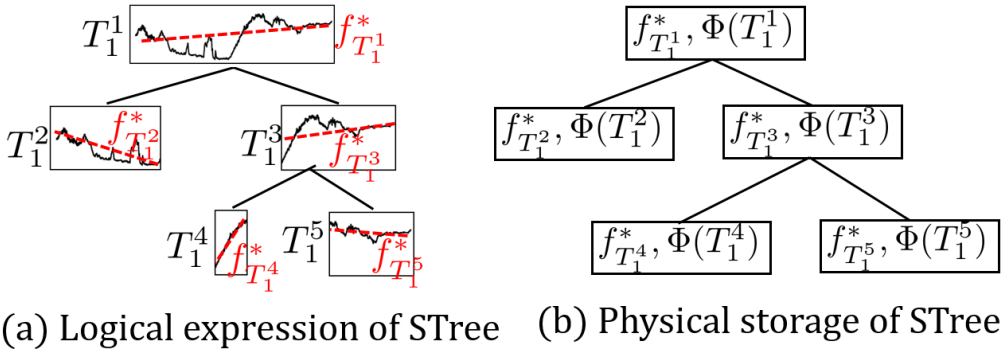


Figure 3.3. Compressed time series segment tree structure.

colors (the corresponding segments are visualized in the right side). [3] *Plato produces the approximate answer* 0.300 *(error is* 0.003*) by using just the function parameters* $(-0.072, 69.38)$*,* $(-0.002, 65.77)$ *for* T_1 *and* $(-0.046, 37.23)$*,* $(-0.038, 38.04)$ *for* T_2*.* [4]

The well-known downside of approximate querying is that errors are introduced. When the example's user receives the approximate answer 0.300 she cannot tell how far this answer is from the *true answer*, i.e., the precise answer. The novelty of Plato is the provision of *tight (i.e., lower bound) deterministic error guarantees* for the answers, even when the time series

[3] How to access the nodes in the segment trees will be fully described in Chapter 3.5.

[4] Due to reasons relating to computation efficiency, as explained in Chapter 3.4.3.2, Plato does not actually store the parameters $(-0.072, 69.38)$, $(-0.002, 65.77)$ and $(-0.046, 37.23)$, $(-0.038, 38.04)$ in their standard basis but rather it stores coefficients in an orthonormal basis.

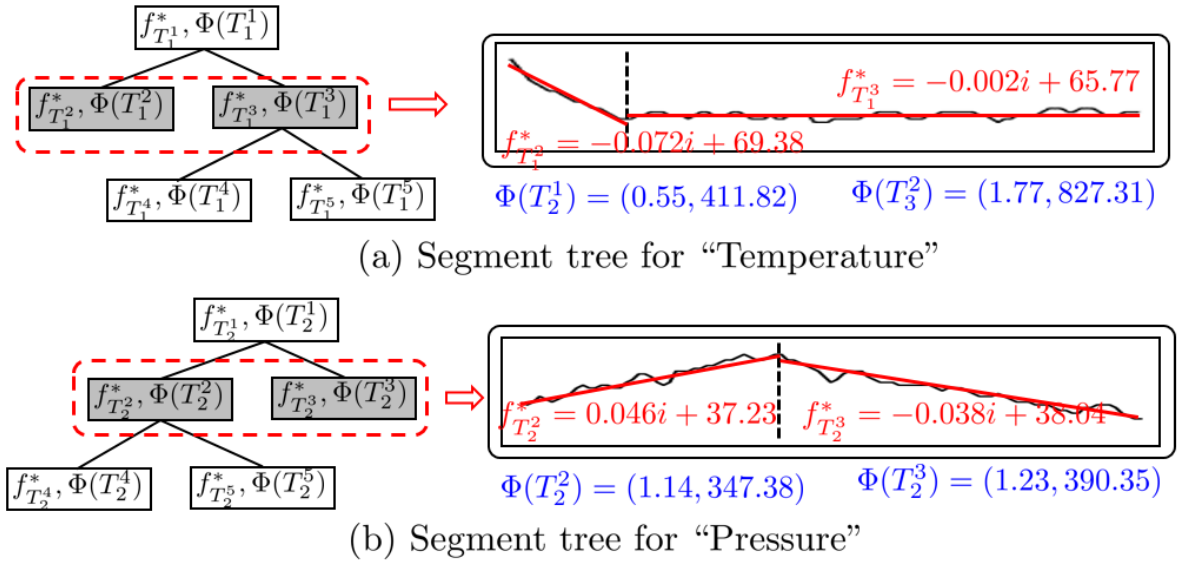


Figure 3.4. Example navigation tree.

expressions combine multiple series. In Example 1, Plato guarantees that the true answer is within ± 0.0032 of the approximate answer 0.300 with 100% confidence. The error guarantee 0.0032 is no greater than the given error budget 0.005. (Indeed, the true answer 0.303 is within ± 0.0032 of 0.300.) It produces these guarantees by utilizing *error measures* associated with each segment.

Scope of Queries and Error Guarantees. Plato supports the time series analytic expressions formally defined in Table 3.4 (Chapter 3.2).They are composed of vector operators ($+$, $-$, $\times$, Shift), arithmetic operators, the aggregation operator Sum that turns its input vector into a scalar, and the Constant operator that turns its input scalar into a vector. As such, Plato queries can express not only statistics that involve one time series (eg, average, variance, and n-th moment) but also statistics that involve multiple time series, such as correlation and cross-correlation.

The error guarantee framework is also general. It allows efficient error guarantee computation for all possible estimation function families, as long as the error measures of Table 3.1 are computed in advance. [5] Figure 3.4 shows the error measures Φ (in blue) for each segment of the

[5] We will show that in certain cases one or two measures suffice.

Table 3.1. Error measures stored for a time series segment T running from a to b and approximated with the estimation function f_T^*.

Error measures	Comments
$\|\varepsilon_T\|_2 = \sqrt{\sum_{i=a}^{b}(T[i] - f_T^*(i))^2}$	L_2-norm of the estimation errors
$\|f_T\|_2 = \sqrt{\sum_{i=a}^{b}(f_T^*(i))^2}$	L_2-norm of the estimated values
$r_T^{\varepsilon} = \|\sum_{i=a}^{b} T[i] - \sum_{i=a}^{b} f_T^*(i)\|$	Absolute reconstruction error

example. With the help of the error measures, no matter whether a time series is compressed by trigonometric functions or polynomial functions or some other family, Plato is able to give tight deterministic error guarantees for queries involving the compressed time series.

Function Family Groups Producing Practical Error Guarantees. Plato produces tight error guarantees, for *any* function family that may have been used in the compression. In addition, our theoretical and experimental analysis identifies which families lead to high quality guarantees.

The formulas of Table 3.2 and Table 3.3 provide error guarantees for characteristic, simple expressions and exhibit the difference in guarantee quality. Any other expression, e.g., the statistics of Table 3.5, are also given error guarantees by composing the error measures and guarantees of their subexpressions (as shown in the paper) and the same quality characterizations apply to them inductively.

This is how to interpret the results of Table 3.2 and Table 3.3: Three function family groups have been identified: (1) The Linear Scalable Family group (LSF), (2) the Vector Space (VS), which includes the LSF and (3) ANY, which, according to its name, includes everything. Given the function family $\mathbb{F}$ used in the compression, we first categorize $\mathbb{F}$ in one of LSF or VS/LSF (i.e., VS excluding LSF) or ANY/VS. For example, if $\mathbb{F}$ is the 2-degree polynomials, then $\mathbb{F}$ belongs to LSF. See Figure 3.5 for other examples. Next, we consider whether the segments of the involved compressed time series are aligned or misaligned and finally we look at the error guarantee formula for the expression.

The specifics of interpreting the table's results and the specifics of their efficient compu-

Table 3.2. Error guarantees for the time series analytic (TSA) $Sum(T_1 \diamond T_2)$ where $\diamond \in \{\times, +, -\}$ on aligned time series compressed by estimation functions in different families.

	function family	error guarantees on aligned time series	AI	Tight
$Sum(\boldsymbol{T_1} \times \boldsymbol{T_2})$	ANY\VS	$\sum_{i=1}^{k} \left(\|\varepsilon_{T_1^i}\|_2 \times \|\varepsilon_{T_2^i}\|_2 \right)$ $+ \sum_{i=1}^{k} \left(\|\varepsilon_{T_1^i}\|_2 \times \|f_{T_2^i}\|_2 \right)$ $+ \sum_{i=1}^{k} \left(\|\varepsilon_{T_2^i}\|_2 \times \|f_{T_1^i}\|_2 \right)$	✗	✓
	VS\LSF	$\sum_{i=1}^{k} \left(\|\varepsilon_{T_1^i}\|_2 \times \|f_{T_2^i}\|_2 \right)$	✓	✓
	LSF			
$Sum(\boldsymbol{T_1} + \boldsymbol{T_2})$	ANY	$\sum_{i=1}^{k} (r^{\varepsilon}_{T_1^i} + r^{\varepsilon}_{T_2^i})$	✓	✓
$Sum(\boldsymbol{T_1} - \boldsymbol{T_2})$				

tation require the detailed discussion of the paper. (Eg, the summation index OPT corresponds to the optimal segment combination (Chapter 3.4.3.1.) Nevertheless, a clear and general high level lesson about the practicality of the error guarantees emerges from the table's summary: *Some function families allow for much higher quality error guarantees than other function families.* The typical characteristic of "higher quality" is *Amplitude Independence (AI)*. If an error guarantee is AI, then it is not influenced by the $\|f_T\|_2$ measure, i.e., it is not affected by the amplitude of the values of the estimation functions and, thus, it is not affected from the amplitude of the original data. An AI error guarantee is only affected by the reconstruction errors caused by the estimation functions, which intuitively implies that AI error guarantees are close to the actual error.

These guarantees are *tight* in the following sense. Given (a) the function family categorization into LSF, VS/LSF or ANY/VS and (b) segments with the error measures of Table 3.1, the formula provided by Table 3.2 and Table 3.3 produces an error guarantee that is as small as

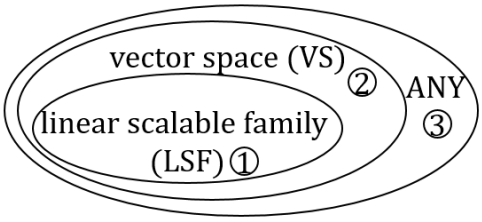
Table 3.3. Error guarantees for the time series analytic (TSA) $Sum(T_1 \diamond T_2)$ where $\diamond \in \{\times, +, -\}$ on misaligned time series compressed by estimation functions in different families. $OPT(L_{T_1}, L_{T_2})$ is the optimal segment combination returned by the algorithm OS in Chapter 3.4.3.1

	function family	error guarantees on misaligned time series	AI	Tight
$Sum(\boldsymbol{T_1} \times \boldsymbol{T_2})$	ANY\VS	$\sum_{i=1}^{k_1} \left(\|\varepsilon_{T_1^i}\|_2 \times (\sum_{j \in \Pi_{T_2,[a_1^i,b_1^i]}} \|f_{T_2^j}\|_2^2)^{\frac{1}{2}} \right)$ $+ \sum_{i=1}^{k_2} \left(\|\varepsilon_{T_2^i}\|_2 \times (\sum_{j \in \Pi_{T_1,[a_2^i,b_2^i]}} \|f_{T_1^j}\|_2^2)^{\frac{1}{2}} \right)$ $+ \sum_{[a,b] \in OPT(L_{T_1}, L_{T_2})} \Big(\big(\sum_{i \in \Pi_{T_1,[a,b]}} \|\varepsilon_{T_1^i}\|_2^2 \big)^{\frac{1}{2}}$ $\times \big(\sum_{i \in \Pi_{T_2,[a,b]}} \|\varepsilon_{T_2^i}\|_2^2 \big)^{\frac{1}{2}} \Big)$	✗	✓
	VS\LSF			
	LSF	$\sum_{i=1}^{k_1} \left(\|\varepsilon_{T_1^i}\|_2 \times \|\boldsymbol{f_{T_2}}\|_{[\boldsymbol{a_1^i}, \boldsymbol{b_1^i}]} - \boldsymbol{f^*_{T_1^i}}\|_2 \right)$ $+ \sum_{i=1}^{k_2} \left(\|\varepsilon_{T_2^i}\|_2 \times \|\boldsymbol{f_{T_1}}\|_{[\boldsymbol{a_2^i}, \boldsymbol{b_2^i}]} - \boldsymbol{f^*_{T_2^i}}\|_2 \right)$ $+ \sum_{[a,b] \in OPT(L_{T_1}, L_{T_2})} \Big(\big(\sum_{i \in \Pi_{T_1,[a,b]}} \|\varepsilon_{T_1^i}\|_2^2 \big)^{\frac{1}{2}}$ $\times \big(\sum_{i \in \Pi_{T_2,[a,b]}} \|\varepsilon_{T_2^i}\|_2^2 \big)^{\frac{1}{2}} \Big)$	✓	✓
$Sum(\boldsymbol{T_1} + \boldsymbol{T_2})$	ANY	$\sum_{i=1}^{k_1} r^{\varepsilon}_{T_1^i} + \sum_{j=1}^{k_2} r^{\varepsilon}_{T_2^j}$	✓	✓
$Sum(\boldsymbol{T_1} - \boldsymbol{T_2})$				

possible. That is, for this superfamily and for the given error measures, any attempt to create a better (i.e., smaller) error guarantee will fail because there are provably time series and at least one time series analytics expression where the true error is exactly as large as the error guarantee.

The experimental results, where we tried data sets with different characteristics and different compression methods, verified the above intuition: AI error guarantees were *order(s) of magnitude smaller* than their amplitude dependent counterparts. Indeed, AI ones over variable-length compressions were invariably small enough to be practically meaningful, while non-AI guarantees were too large to be practically useful.

Particularly interesting are the analytics that combine multiple vectors, such as correlation and cross-correlation, by vector multiplication. Then the amplitude independence of the error guarantees does not apply generally. Rather the dichotomy illustrated in Figure 3.6 emerges:

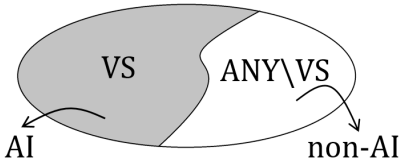


(a) Function family groups

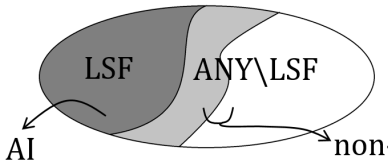
Family Group	Example Function Family
LSF①	$\{ax + b \mid a, b \in R\}$
VS\LSF②	$\{a \ln(1 + x) + b \mid a, b \in R\}$
ANY\VS③	$\{\frac{L}{1+\exp(ax+b)} + c \mid L, a, b, c \in R\}$
	$\{a \exp(\frac{-(x-b)^2}{2c^2}) + d \mid a, b, c, d \in R\}$

(b) Example function families

Figure 3.5. Function family groups and examples.



(a) Function family dichotomy for error guarantees of queries on aligned time series



(b) Function family dichotomy for error guarantees of queries on misaligned time series

Figure 3.6. Function family groups and resulting guarantees

(i) for compressions with aligned time series segments, the error guarantee is AI when the used function family forms a *Vector Space (VS)* in the conventional sense [36]; and (ii) for compressions with misaligned time series segments, which are the more common case, choosing a VS family is not enough for AI guarantees. The family must be a *Linear Scalable Family (LSF)*, which is a property that we define in this paper (Chapter 3.3.1.1).

The contributions are summarized as follows.

- We deliver tight deterministic error guarantees for a wide class of analytics over compressed time series. The key challenge is analytics (e.g., correlation and cross-correlation) that combine multiple time series but it is not known in advance which time series may be combined. Thus, each time series has been compressed individually, much before a query arrives. The reconstruction errors of the individual time series' compressions cannot provide, by themselves, decent guarantees for queries that multiply time series. To make the problem harder, time series segmentations are generally misaligned.[6]

[6]Misalignment happens because the most effective compressions use variable length segmentations. But even if

- The provided guarantees apply regardless of the specifics of the segmentation and estimation function family used during the compression, thus making the provided deterministic error guarantees applicable to any prior work on segment-based compression (eg, variable-sized histograms etc). The only requirement is the common assumption that the estimation function minimizes the Euclidean distance between the actual values and the estimates.

- We identify broad estimation function family groups (namely, the already defined Vector Space family and the presently defined Linear Scalable Family) that lead to theoretically and practically high quality guarantees. The theoretical aspect of high quality is crisply captured by the Amplitude Independence (AI) property. Furthermore, the error guarantees are computed very efficiently, in time proportional to the number of segments.

- The results broadly apply to analytics involving composition of the typical operators, which is powerful enough to express common statistics, such as variance, correlation, cross-correlation and other in any time range.

- We propose a native compressed segment tree structure and a top-down tree navigation algorithm to support analytic queries with error/time budgets. In addition, we propose an incremental estimation function computation method to significantly improve the tree building performance and also an incremental error guarantee computation optimization to improve the error guarantee computation performance.

- We conduct an extensive empirical evaluation on four real-life datasets to evaluate the error guarantees provided by Plato and the importance of the VS and LSF properties on error estimation. The results show that the AI error guarantees are very narrow - thus, practical. Furthermore, we compare to sampling-based approximation and show experimentally that Plato delivers deterministic (100% confidence) error guarantees using fewer data than it takes to produce probabilistic error guarantees with 95% and 99% confidence via sampling.

the segmentations were fixed length, queries such as cross-correlation and cross-auto-correlation time shift one of their time series, thus producing misalignment with the second time series.

Table 3.4. Grammar of time series analytic (TSA). Let $\boldsymbol{T}_1 = (a_1, b_1, [\boldsymbol{T_1}[a_1], ..., \boldsymbol{T_1}[b_1]])$ and $\boldsymbol{T}_2 = (a_2, b_2, [\boldsymbol{T_2}[a_2], ..., \boldsymbol{T_2}[b_2]])$ be the input time series in the time series expressions, $a = \max(a_1, a_2)$ and $b = \min(b_1, b_2)$

Time Series Analytic (TSA)			
Q	$\rightarrow$	Ar	
Arithmetic Expression (Ar)			
Ar	$\rightarrow$	literal value in R	where $\otimes \in \{+, -, \times, \div, \sqrt{}\}$
	—	Ar $\otimes$ Ar	
	—	Agg	
Aggregation Expression (Agg)			
Agg	$\rightarrow$	Sum($\boldsymbol{T}, a', b'$)	$\sum_{i=a'}^{b'} \boldsymbol{T}[i]$, where $[a', b'] \subseteq [a, b]$
Time Series Expression (TSE)			
$\boldsymbol{T}$	$\rightarrow$	input time series	
	—	Serialize(v, a, b)	$(a, b, [\underbrace{v, v, ..., v}_{b-a+1}])$
	—	Shift($\boldsymbol{T}, k$)	$(a + k, b + k, [\boldsymbol{T}[a], ..., \boldsymbol{T}[b]])$
	—	$\boldsymbol{T_1} + \boldsymbol{T_2}$	$(a, b, [\boldsymbol{T_1}[a] + \boldsymbol{T_2}[a], ..., \boldsymbol{T_1}[b] + \boldsymbol{T_2}[b]])$
	—	$\boldsymbol{T_1} - \boldsymbol{T_2}$	$(a, b, [\boldsymbol{T_1}[a] - \boldsymbol{T_2}[a], ..., \boldsymbol{T_1}[b] - \boldsymbol{T_2}[b]])$
	—	$\boldsymbol{T_1} \times \boldsymbol{T_2}$	$(a, b, [\boldsymbol{T_1}[a] \times \boldsymbol{T_2}[a], ..., \boldsymbol{T_1}[b] \times \boldsymbol{T_2}[b]])$

3.2 Time Series and Expressions

Time Series. A time series $\boldsymbol{T} = (a, b, [\boldsymbol{T}[a], \boldsymbol{T}[a+1], ..., \boldsymbol{T}[b]])$, $a \in N$, $b \in N$, is a sequence of data points $[\boldsymbol{T}[a], \boldsymbol{T}[a+1], ..., \boldsymbol{T}[b]]$ observed from start time a to end time b ($a, b \in N$). Following the assumptions in [69, 20, 98] we assume that time is discrete and the resolution of any two time series is the same. Equivalently, we say $\boldsymbol{T}$ is fully defined in the integer time domain $[a, b]$. We assume a domain $[1, n]$ is the global domain meaning that all the time series are defined within subsets of this domain. When the domain of a time series $\boldsymbol{T}$ is implied by the context, then $\boldsymbol{T}$ can be simplified as $\boldsymbol{T} = [\boldsymbol{T}[a], \boldsymbol{T}[a+1], ..., \boldsymbol{T}[b]]$.

Example 2. *Assume the global domain is* $[1, 100]$. *Consider two time series* $\boldsymbol{T}_1 = (1, 5, [61.52, 59.54, 58.64, 59.36, 60.44])$ *and* $\boldsymbol{T}_2 = (3, 6, [1.02, 1.03, 1.02, 1.02])$. *Then* $\boldsymbol{T}_1$ *and* $\boldsymbol{T}_2$ *are fully defined in domains [1,5] and [3,6] respectively.* $\boldsymbol{T}_2[4] = 1.03$ *refers to the* 2^{nd} *data point of* $\boldsymbol{T}_2$

Table 3.5. Example TSA's for common statistics. Let $\boldsymbol{T}_1 = (a_1, b_1, [\cdots])$ and $\boldsymbol{T}_2 = (a_2, b_2, [\cdots])$ be the input time series in the time series analytic

TSA Expression	Equivalent TSA Expression	Usage of error measures
Average μ_{T_1} '$\mu(T_1)$'	$\frac{1}{b_1 - a_1 + 1}(\text{Sum}(\boldsymbol{T_1}))$	$r^{\varepsilon}_{T_1}$
Standard Deviation σ_{T_1} '$\sigma(T_1)$'	$\sqrt{\frac{1}{b_1 - a_1 + 1} \times Sum(\boldsymbol{T_1} - \mathsf{Serialize}(\mu_{T_1}))}$	$r^{\varepsilon}_{T_1}$
Correlation $r_{(T_1,T_2)}$ 'Corr(T_1,T_2)'	$\frac{Sum\Big((\boldsymbol{T_1} - \mathsf{Serialize}(\mu_{T_1})) \times (\boldsymbol{T_2} - \mathsf{Serialize}(\mu_{T_2}))\Big)}{\sigma_{T_1} \times \sigma_{T_2}}$	$\|\varepsilon_{T_1}\|_2$,$\|f_{T_1}\|_2$,$r^{\varepsilon}_{T_1}$, $\|\varepsilon_{T_2}\|_2$,$\|f_{T_2}\|_2$,$r^{\varepsilon}_{T_2}$
Cross-correlation $r_{(T_1,T_2,m)}$ 'CCorr(T_1,T_2,m)'	$\frac{Sum\Big((\boldsymbol{T_1} - \mathsf{Serialize}(\mu_{T_1})) \times (Shift(\boldsymbol{T_2}, m) - \mathsf{Serialize}(\mu_{T_2}))\Big)}{\sigma_{T_1} \times \sigma_{T_2}}$	$\|\varepsilon_{T_1}\|_2$,$\|f_{T_1}\|_2$,$r^{\varepsilon}_{T_1}$, $\|\varepsilon_{T_2}\|_2$,$\|f_{T_2}\|_2$,$r^{\varepsilon}_{T_2}$
Auto-correlation $r_{(T_1,m)}$ 'ACorr(T_1,m)'	$\frac{Sum\Big((\boldsymbol{T_1} - \mathsf{Serialize}(\mu_{T_1})) \times (Shift(\boldsymbol{T_1}, m) - \mathsf{Serialize}(\mu_{T_1}))\Big)}{\sigma_{T_1} \times \sigma_{T_1}}$	$\|\varepsilon_{T_1}\|_2$,$\|f_{T_1}\|_2$,$r^{\varepsilon}_{T_1}$

at the 4*-th position in the global domain.*

Time Series Analytic (TSA) Expressions. Table 3.4 shows the formal definition of the *time series analytic* (called *TSA*). The TSAs supported are expressions composed of linear algebra operators and arithmetic operators. Typically, the TSA has subexpressions that compose one or more linear algebra operators over multiple time series vectors as defined below.

- Given a numeric value v and two integers a and b, $\mathsf{Serialize}(v, a, b) = (a, b, [v, ..., v])$. For example, $\mathsf{Serialize}(1.6, 3, 5)$ produces $(3, 5, [1.6, 1.6, 1.6])$.

- Given a time series $\boldsymbol{T} = (a, b, [\boldsymbol{T}[a], ..., \boldsymbol{T}[b]])$ and an integer value k, $\mathsf{Shift}(\boldsymbol{T}, k)$=$(a + k, b + k, [\boldsymbol{T}[a], ..., \boldsymbol{T}[b]])$. Notice $\mathsf{Shift}(\boldsymbol{T}, k)[i + k] = \boldsymbol{T}[i]$ for all $a \leq i \leq b$. Figure 3.7(a) visualizes the Shift operator. Consider the time series $\boldsymbol{T} = (1, 3, [1.8, 1.6, 1.6])$, then $\mathsf{Shift}(\boldsymbol{T}, 6)$ is $(7, 9, [1.8, 1.6, 1.6])$.

- Given two time series $\boldsymbol{T}_1 = (a_1, b_1, [\boldsymbol{T}_1[a_1], ..., \boldsymbol{T}_1[b_1]])$ and $\boldsymbol{T}_2 = (a_2, b_2, [\boldsymbol{T}_2[a_2], ..., \boldsymbol{T}_2[b_2]])$, $\boldsymbol{T_1} \times \boldsymbol{T_2} = (a, b, [\boldsymbol{T}_1[a] \times \boldsymbol{T}_2[a], ..., \boldsymbol{T}_1[b] \times \boldsymbol{T}_2[b]])$ where $a = max(a_1, a_2)$ and $b = min(b_1, b_2)$. [7]

[7] Setting $a = max(a_1, a_2)$ and $b = min(b_1, b_2)$ ensures all the data points in $\boldsymbol{T_1} \times \boldsymbol{T_2}$ are defined.

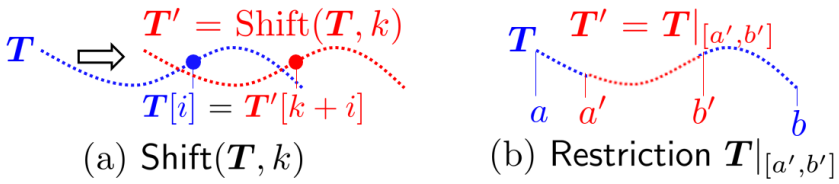


Figure 3.7. Time series *Shift* and *Restriction* operators.

For example, given $\boldsymbol{T}_1 = (1, 2, [3.3,\ 3.5])$ and $\boldsymbol{T}_2 = (1, 2, [1.0, 1.2])$ then $\boldsymbol{T_1} \times \boldsymbol{T_2} = (1, 2, [3.3, 4.2])$. Similarly, we define $\boldsymbol{T_1} + \boldsymbol{T_2}$ and $\boldsymbol{T_1} - \boldsymbol{T_2}$.

A time series analytic (TSA) is an arithmetic expression of the form $Arr_1 \otimes Arr_2 \otimes \ldots Arr_n$, where $\otimes$ are the standard arithmetic operators $(+, -, \times, \div, \sqrt{})$ and Arr_i is either an arithmetic literal or an aggregation over a time series expression. An aggregation expression $\mathsf{Sum}(\boldsymbol{T}, a', b')$ computes the summation of the data points of T in the domain $[a', b']$, i.e., $\mathsf{Sum}(\boldsymbol{T}, a', b')$=$\sum_{i=a'}^{b'} \boldsymbol{T}[i]$ where $\boldsymbol{T}$ can be an input time series or a derived time series computed by time series expressions (TSEs). [8] When the bounds of a' and b' are implied from the context, we simplify $Sum(\boldsymbol{T}, a', b')$ to $Sum(\boldsymbol{T})$.

3.3 Internal, Compressed Time Series Segment Tree

When a user inserts a time series into the database, Plato physically stores the *compressed time series segment tree* (called *STree*) instead of the raw time series. More precisely, the user provides (i) a time series T, (ii) the maximal error bound θ, which restricts the maximal error produced in each segment, and (iii) the identifier of a function family, which is selected from a list provided by Plato. Internally, Plato calls the tree building algorithm to build an STree for T. Each node stores a *compressed segment representation* $\tilde{T}^i = (a, b, \tilde{f}_T^*, \Phi(T))$ of the segment T, where a is the start position, b is the end position, $\tilde{f}_T^*$ is the function representation of f_T^*, where f_T^* is the estimation function chosen from the identified function family and $\Phi(T)$ is a set of (one to three depending on the function family) *error measures*.

[8] Note that, when the time series expressions involve time shifting, we assume that the aggregation will only operator in the valid data points, that is the data points in the defined range.

Overall, for a time series T, Plato physically stores (i) a compressed segment tree, and (ii) one token (which can simply be an integer) as the function family identifier.

Next, we introduce the structure of STree in Chapter 3.3.1. Then we present the segment tree building algorithm in Chapter 3.3.2.

3.3.1 Compressed Segment Tree Structure

Plato builds one compressed segment tree structure (STree) for each time series. STree has the following properties:

- Each node in an STree logically represents a time series segment T and physically stores the *compressed segment representation* $\tilde{T^i} = (a, b, \tilde{f_T^*}, \Phi(T))$, where a is the start position, b is the end position, $\tilde{f_T^*}$ is the function representation of f_T^*, where f_T^* is the estimation function chosen from the identified function family and $\Phi(T)$ is a set of (one to three depending on the function family) *error measures*.

- An STree may not be a balanced tree. Some segments of a time series may be relatively smooth and well behaved. Such segments can be easily compressed with very little estimation error. On the other hand, some segments may be quite ill-behaved. These segments require us to partition the data into several sub-segments in order to reduce the estimation error. Motivated by the above observations, STree uses more nodes on the ill-behaved segments while fewer nodes on the well-behaved segments, which leads to an unbalanced tree structure.

Figure 3.3 provides an example of an STree constructed with 1-degree polynomial estimation functions.

In the following, we introduce the estimation function selection (Chapter 3.3.1.1) and error measures computation (Chapter 3.3.1.2).

3.3.1.1 Estimation Function Selection

Choosing an estimation function for a time series segment has two steps: (i) user identifies the function family, and (ii) Plato selects the best function in the family, i.e., the function that

Table 3.6. Example function family identifiers

τ	Expression	Comment
p_i	$\{\sum_{i=0}^{i} a_i x^i \| a_i \in R\}$	i-degree Polynomial
g	$\{a \exp(\frac{-(x-b)^2}{2c^2}) + d \| a, b, c, d \in R\}$	Gaussian
l	$\{\frac{L}{1+\exp(ax+b)} + c \| L, a, b \in R\}$	Logistic

minimizes the Euclidean distance between the original values and the estimated values produced by the function.

Step 1: Function family selection. Table 3.6 gives example function family identifiers, which the user may select, and the corresponding function expressions. For example, τ="p_2" means that the chosen function family is the "second-degree polynomial function family" and the corresponding function family expression is $\{ax^2 + bx + c | a, b, c \in R\}$.

Step 2: Estimation function selection. Any function f in the chosen function family $\mathbb{F}$ is a *candidate estimation function*. Following the prior work [60, 6], Plato selects the candidate estimation function that minimizes the Euclidean distance between the original values and the estimated values produced by the function to be the final estimation function. More precisely,

$$f_T^* = \arg\min_{f \in \mathbb{F}} \Big(\sum_{i=a}^{b} (\boldsymbol{T}[i] - f(i))^2 \Big)^{1/2} \tag{3.1}$$

Example 3. *Given a time series* $\boldsymbol{T} = (1, 5, [0.2, 0.4, 0.4, 0.5, 0.6])$*, assume the function family identifier is "*p_1*" (i.e., "first-degree polynomial function family"). Functions* $f_1 = 0.05 \times i + 0.3$ *and* $f_2 = 0.09 \times i + 0.15$ *are two candidate estimation functions. Finally, Plato selects* $f_2 = 0.09 \times i + 0.15$ *as the estimation function since it produces the minimal Euclidean error, i.e.,* 0.0837.

Function Representation (Physical) vs. Function (Logical). Once an estimation function f_T^* is selected, Plato stores the corresponding *function representation* $\tilde{f_T^*}$, which includes (i) the coefficients of the function f_T^*, and (ii) the function family identifier τ. [9] For example, the

[9] All the segments in the same time series share one token τ.

function representation of the estimation function in Example 3 is $\tilde{f}_T^*$ = $((0.09, 0.15)$, $p_1)$ where p_1 is a function family identifier indicating that the function family is "1-degree polynomial function family".

When we talk about the function itself logically, it can be regarded as a vector that maps time series: given a domain $[a, b]$, the vector $[f(a), f(a+1), \ldots, f(b)]$ maps a value to each position in the domain $[a, b]$. For example, consider the estimation function $f_T^* = 0.09 \times i + 0.15$ in Example 3. Then $\boldsymbol{T} - f_T^* = [0.2 - f_T^*(1), 0.4 - f_T^*(2), 0.4 - f_T^*(3), 0.5 - f_T^*(4), 0.6 - f_T^*(5)] = [0.2 - 0.24, 0.4 - 0.33, 0.4 - 0.42, 0.5 - 0.51, 0.6 - 0.6] = [0.04, 0.07, -0.02, -0.01, 0]$.

Coefficients in an orthonormal basis. For a finite vector space (including the LSF), we physically store the coefficients of the estimation function in an orthonormal basis. The benefit of using coefficients in an orthonormal basis is that it supports fast computation of L_2-norm of the estimation functions, which is important in providing error guarantees for queries involving misaligned time series.

Let $(\varphi_1^T, ..., \varphi_{dim(\mathbb{F}_T)}^T)$ be an orthonormal basis of $\mathbb{F}$ for the scalar product $\langle f_1, f_2 \rangle = \sum_{i=a}^{b} f_1(i) \times f_2(i)$, where $dim(\mathbb{F})$ is the dimension of $\mathbb{F}_T$. For example, if $\mathbb{F}$ is 1-degree polynomial function family, then $\mathbb{F} = 2$. Notice $dim(\mathbb{F})$ is much smaller than $|T|$, otherwise the original data points can be directly stored instead of using functions. Such orthonormal basis can be computed by using Gram-Schmidt process [9]. Example 4 shows (i) how to compute the orthonormal basis, and (ii) how to transform coefficients from the standard basis to the orthonormal basis. Notice that, the complexities of step (i) and (ii) are $O(dim(\mathbb{F}))$.

Example 4 (Orthonormal basis)**.** *Consider the 1-degree polynomial function family and a domain [1,m+1], then the standard basis is* $[\vec{v_1}, \vec{v_2}] = [\vec{1}, \vec{X}]$. *Based on the GramSchmidt process,* $[\vec{v_1}, \vec{v_2}]$ *can be converted to orthogonal vectors* $[\vec{u_1}, \vec{u_2}]$ *as follows:*

$$\vec{u_1} = \vec{v_1} = \vec{1}$$

$$\vec{u_2} = \vec{v_2} - \frac{\langle \vec{u_1}, \vec{v_2} \rangle}{\langle \vec{u_1}, \vec{u_1} \rangle} \vec{u_1} = \vec{X} - \frac{\langle \vec{1}, \vec{X} \rangle}{\langle \vec{1}, \vec{1} \rangle} \vec{1} = \vec{X} - \frac{m+1}{2} \vec{1}$$

Then $[\vec{u_1}, \vec{u_2}]$ can be normalized to orthonormal basis $[\phi_1, \phi_2]$ as follows:

$$\phi_1 = \frac{\vec{u_1}}{\|\vec{u_1}\|} = \frac{\vec{1}}{\|\vec{1}\|} = \frac{\vec{1}}{\sqrt{m+1}}$$
$$\phi_2 = \frac{\vec{u_2}}{\|\vec{u_2}\|} = \frac{\vec{X} - \frac{m+1}{2}\vec{1}}{\|\vec{X} - \frac{m+1}{2}\vec{1}\|} = \frac{\vec{X} - \frac{m+1}{2}\vec{1}}{\sqrt{\frac{m+1}{12}((m+1)^2 - 1)}}$$

Consider the time series segment $T|_{[1,9]}$ with an estimation function $f_T^* = 1.2 + 0.1x$. The coefficients in the standard basis are as follows:

$$\begin{bmatrix} c_0 \\ \\ c_1 \end{bmatrix} = \begin{bmatrix} 1.2 \\ \\ 0.1 \end{bmatrix}$$

In order to obtain the corresponding coefficients $[c_0', c_1']$ in the orthonormal basis, we first need to compute a change-of-basis matrix M mapping the standard basis to the orthonormal basis. The M can be computed from the following two equations.

$$\vec{1} = a_0\phi_0 + a_1\phi_1$$
$$\vec{x} = b_0\phi_0 + b_1\phi_1$$

From here we derive the change-of-basis matrix M:

$$M = \begin{bmatrix} a_0 & b_0 \\ \\ a_1 & b_1 \end{bmatrix} = \begin{bmatrix} \sqrt{n} & \frac{\sqrt{n}(n+1)}{2} \\ \\ 0 & \sqrt{\frac{n}{12}(n^2-1)} \end{bmatrix}$$

In this example, we have $n = 9$, so the final M is:

$$M = \begin{bmatrix} 3 & 6 \\ \\ 0 & \sqrt{60} \end{bmatrix}$$

Then the coefficients $[c'_0, c'_1]$ in orthonomal basis can be derived in the following formula:

$$\begin{bmatrix} c'_0 \\ c'_1 \end{bmatrix} = M \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} 3 & 6 \\ 0 & \sqrt{60} \end{bmatrix} \begin{bmatrix} 1.2 \\ 0.1 \end{bmatrix} = \begin{bmatrix} 4.2 \\ 0.2\sqrt{15} \end{bmatrix}$$

Notice that, the coefficients $[c'_0, c'_1]$ in orthonomal basis can be converted back to the $[c_0, c_1]$ in the standard basis by computing

$$\begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = M^{-1} \begin{bmatrix} c'_0 \\ c'_1 \end{bmatrix}$$

3.3.1.2 Error Measures

In addition to the estimation function, Plato stores extra *error measures* $\Phi(T) = \{\|\varepsilon_T\|_2, \|f_T\|_2, r_T^{\varepsilon}\}$ for each time series segment T (defined in domain $[a, b]$) where $\|\varepsilon_T\|_2$, $\|f_T\|_2$, and r_T^{ε} are defined in Table 3.1.

Example 5. *Consider the time series $\boldsymbol{T} = (1, 5, [0.2, 0.4, 0.4, 0.5, 0.6])$ in Example 3 again. $f_T^* = 0.09 \times i + 0.15$ is the estimation function. Thus $\|\varepsilon_T\|_2 = \sqrt{\sum_{i=1}^{5}(T[i] - f_T^*(i))^2} = 0.0837$, $\|f_T\|_2 = \sqrt{\sum_{i=1}^{5}(f_T^*(i))^2} = 0.9813$, and $r_T^{\varepsilon} = |\sum_{i=1}^{5} T[i] - \sum_{i=1}^{5} f_T^*(i)| = 2.1 - 2.1 = 0$.*

Elimination of r_T^{ε}. We will see in Lemma 9 (Chapter 3.4.2.1) that if the selected function family forms a vector space, then r_T^{ε} is guaranteed to be 0. Then we can avoid storing it.

3.3.2 Compressed Segment Tree Building Algorithm

We first introduce the segment tree building algorithm *STreeBuilder* in Chapter 3.3.2.1, then we improve the performance of *STreeBuilder* by applying the optimization of the incremental estimation function computation (Chapter 3.3.2.2).

Algorithm: STreeBuilder

Input: Time series T with domain $[a, b]$ and maximal error bound θ
Output: STree $root$

1 $f_T^* \leftarrow$ *ComputeEstimationFunction*(T);
2 $root \leftarrow$*CreateNode*(f_T^*, T);
3 $\varepsilon \leftarrow$*ComputeError*(f_T^*, T);
4 **if** $\varepsilon > \theta$ **then**
5 $\quad p \leftarrow$*FindBestPosition*(T);
6 $\quad (root \rightarrow left) \leftarrow$*STreeBuilder*($T|_{[a,p]}, \theta$);
7 $\quad (root \rightarrow right) \leftarrow$*STreeBuilder*($T|_{[p+1,b]}, \theta$);
8 Return $root$;

3.3.2.1 Top-down Building Algorithm

The segment tree building algorithm *STreeBuilder* takes a time series T and a maximal error bound θ as input and outputs a compressed segment tree structure STree. STreeBuilder is a top-down building algorithm, which works as follows:

1. STreeBuilder first compresses time series T using the best fitting estimation function f_T^* (line 1). Then it creates the root node for T (line 2) and calculates the estimation error ε (line 3).

2. If ε is greater than the given error bound θ (line 4), then it finds the best cutting position p (line 5), and recursively calls STreeBuilder to build the left subtree for $T|_{[a,p]}$ (line 6) and the right subtree for $T|_{[p+1,b]}$ (line 7). Otherwise, it outputs $root$ (line 8).

Figure 3.3 is an example STree built by the STreeBuilder algorithm.

Best Cutting Position Selection. Basically, the best cutting position p is the position leading to the minimum total estimation errors. Consider a time series segment $T|_{[a,b]}$, then point $p \in [a, b]$ is the best cutting position if $\nexists c \in [a, b]$ s.t. $(\|\varepsilon_{T|_{[a,c]}}\|_2 + \|\varepsilon_{T|_{[c+1,b]}}\|_2) < (\|\varepsilon_{T|_{[a,p]}}\|_2 + \|\varepsilon_{T|_{[p+1,b]}}\|_2)$.

Let N be the number of data points in time series segment T. The function *FindBest-Position* [10] needs to compute $2N$ estimation functions in order to find the best cutting position.

[10]The complexity of STreeBuilder algorithm is dominated by the function FindBestPosition.

And the cost of each computation of estimation function is $O(N)$. Therefore, the total time complexity of function *FindBestPosition* is $O(N^2)$.

3.3.2.2 Incremental Building Algorithm

Recall that the complexity of function *FindBestPosition* is $O(N^2)$ due to the fact that we need to compute the estimation function by using all the data points in the segment, which is time-consuming. For example, building an STree for a time series with 1 million data points by Algorithm takes more than 10 hours.

In order to improve the performance, we employ an *incremental estimation function computation* optimization, which can reduce the complexity of function *FindBestPosition* from $O(N^2)$ to $O(N \times dim(\mathbb{F}))$. The main idea of the incremental estimation function computation is that we can compute the estimation function of $T_{[1,m+1]}$ without performing the entire regression process over all the data points from 1 to $m+1$, but just by reusing the estimation function of $T|_{[1,m]}$. More precisely, $f^*_{T|_{[1,m+1]}}$ is computed as follows:

$$f^*_{T|_{[1,m+1]}} = f^*_{T|_{[1,m]}} + \Delta_f$$

where Δ_f can be computed in $O(dim(F))$, which will be introduced next.

Before jumping deeper into the incremental update optimization, we highlight the power of the optimization here: with the optimized algorithm, the segment index building algorithm can achieve two-orders of magnitudes speedup. For example, building the STree for the time series with 1 million data points now takes less than 1 minutes.

Computation of Δ_f. The key challenge of the incremental update optimization is to compute Δ_f in $O(dim(F))$. We employ the orthonormal basis to achieve this goal. More precisely, Δ_f can be calculated as follows:

$$\Delta_f = \sum_{i=1}^{dim(F)} \Delta_{c_i} \phi_i^{[m+1]} = \sum_{i=1}^{dim(F)} (T[m+1] - f^*_T(m+1))(\phi_i^{[m+1]}(m+1))\phi_i^{[m+1]}$$

where ϕ_i is the i-th vector in the orthonormal basis respect to the inner product space, which can be converted from the normal basis via the GramSchmidt process [9] in $O(dim(F))$.

Example 6 shows how to compute Δ_f in O(dim(F)) with the help of the orthonormal basis.

Example 6 (Computation of Δ_f)**.** *Assume the function family is chosen to be the 1-degree polynomial function family, consider a segment $T|_{[1,8]}$ with an estimation function $f^*_T = 0.1x+1.2$, then the estimation function of the segment $T' = T|_{[1,9]}$ with one new data point $T[9] = 2.6$ can be computed as $f^*_{T'} = f^*_T + \Delta_f = (0.1x + 1.2) + \Delta_f$. To compute Δ_f, we have:*

$$\begin{aligned}
\Delta_f &= \sum_{i=1}^{2} c_i \phi_i^{[9]} = \sum_{i=1}^{2} (T[9] - f^*_T(9))(\phi_i^{[9]}(9))\phi_i^{[9]} \\
&= (2.6 - 2.1) \times \phi_1^{[9]}(9) \times \phi_1^{[9]} + (2.6 - 2.1) \times \phi_2^{[9]}(9) \times \phi_2^{[9]} \\
&= \frac{1}{2} \times \frac{1}{3} \times \frac{\vec{1}}{3} + \frac{1}{2} \times \frac{2}{\sqrt{15}} \times \frac{\vec{X} - \frac{9}{2}\vec{1}}{\sqrt{60}} \\
&= \frac{\vec{X}}{30} - \frac{\vec{1}}{8\sqrt{15}}
\end{aligned}$$

Finally, $f^*_{T'} = f^*_T + \Delta_f = (0.1x + 1.2) + \Delta_f = (0.1 + \frac{1}{30})x + (1.2 - \frac{1}{8\sqrt{15}})$.

3.4 Error Guarantee Computation on Compressed Segment Lists

Before directly jumping into computing the error guarantees on compressed segment trees, in this chapter, we first introduce the simpler case, i.e., computing the error guarantees on compressed segment lists, which serves as the fundamental operation in computing error guarantees on compressed segment trees.

Error Guarantee Definition. Given a TSA q involving time series $T_1, .., T_n$, let R be the accurate answer of q by executing q directly on the original data points of $T_1, .., T_n$. Let $\hat{R}$ be the approximate answer of q by executing q on the compressed time series representations. Then

$\varepsilon = |\hat{R} - R|$ is the *true error* of q. Notice that ε is unknown since R is unknown. An upper bound $\hat{\varepsilon}$ ($\hat{\varepsilon} \geq \varepsilon$) of the true error is called a *deterministic error guarantee* of q. With the help of $\hat{\varepsilon}$, we know that the accurate answer R is within the range $[\hat{R} - \hat{\varepsilon}, \hat{R} + \hat{\varepsilon}]$ with 100% confidence. Plato provides *tight* deterministic error guarantees for time series expressions defined in Table 3.4 (Chapter 3.2).

Error Guarantee Decomposition. Recall that the time series analytic q defined in Table 3.4 (Chapter 3.2) combines one or more time series aggregation operations via arithmetic operators, i.e., $q = Agg_1 \otimes Agg_2 \otimes \cdots \otimes Agg_n$ where $\otimes \in \{+, -, \times, \div, \sqrt{}\}$. In order to provide the deterministic error guarantee $\hat{\varepsilon}$ of the time series analytic q, the key step is to calculate the deterministic error guarantee $\hat{\varepsilon}_{Agg_i}$ of each aggregation operation Agg_i. Once we have $\hat{\varepsilon}_{Agg_i}$ for each aggregate expression, it is not hard to combine them to get the final error guarantee. For arithmetic operator $Ar_1 \otimes\ Ar_2$ where $\otimes\{+, -, \times, \div\}$. If both Ar_1 and Ar_2 are scalar values, the Plato gives accurate answers. Then we discuss in the following two cases: (i) Ar_1 or Ar_2 is an aggregation result produced by Plato, and (ii) both Ar_1 and Ar_2 are aggregation results produced by Plato.

- Case 1. Without loss of generality, we assume Ar_1 is an aggregation operator and Ar_2 is a scalar value. Let $\hat{R}$ be the approximate answer provided by Plato for Ar_1 and $\hat{\varepsilon}$ is the corresponding error guarantee. The approximate answer and the error guarantee of $Ar_1 \otimes\ Ar_2$ is summarized in Table 3.7.

- Case 2. Both Ar_1 and Ar_2 are aggregation operators. Let $\hat{R}_1$ (resp. $\hat{R}_2$) and $\hat{\varepsilon}_1$ (resp. $\hat{\varepsilon}_2$) be the approximate answer and error guarantee provided by Plato for Ar_1 and Ar_2 respectively. The approximate answer and the error guarantee of $Ar_1 \otimes\ Ar_2$ is summarized in Table 3.8.

Given a TSA $Agg = \mathsf{Sum}(\boldsymbol{T})$ and the compressed time series representation $L_T = \{\tilde{T^1}, ..., \tilde{T^k}\}$. When calculating $\hat{\varepsilon}_{Agg}$, there are two cases depending on whether $\boldsymbol{T}$ is an input time series or not. [11]

[11] If a time series is generated by applying some time series operators, then it is not a base time series. For example, $\boldsymbol{T} = \boldsymbol{T_1} \times \boldsymbol{T_2}$, then $\boldsymbol{T}$ is not a base time series.

Table 3.7. Error guarantee propagation in case 1

Operator	approximate answer	error guarantee
$Ar_1 + Ar_2$	$\hat{R} + Ar_2$	$\hat{\varepsilon}$
$Ar_1 - Ar_2$	$\hat{R} - Ar_2$	$\hat{\varepsilon}$
$Ar_1 \times Ar_2$	$\hat{R} \times Ar_2$	$\hat{\varepsilon} \times Ar_2$
$Ar_1 \div Ar_2$	$\hat{R} \div Ar_2$	$\hat{\varepsilon} \div Ar_2$

Table 3.8. Error guarantee propagation in case 2

Operator	approximate answer	error guarantee
$Ar_1 + Ar_2$	$\hat{R}_1 + \hat{R}_2$	$\hat{\varepsilon}_1 + \hat{\varepsilon}_2$
$Ar_1 - Ar_2$	$\hat{R}_1 - \hat{R}_2$	$\hat{\varepsilon}_1 + \hat{\varepsilon}_2$
$Ar_1 \times Ar_2$	$\hat{R}_1 \times \hat{R}_2$	$\hat{\varepsilon}_1\hat{R}_2 + \hat{\varepsilon}_2\hat{R}_1 + \hat{R}_1\hat{R}_2$
$Ar_1 \div Ar_2$	$\hat{R}_1 \div \hat{R}_2$	$\frac{(\hat{\varepsilon}_1\hat{R}_2 + \hat{\varepsilon}_2\hat{R}_1)}{(\hat{R}_2 - \hat{\varepsilon}_2)\hat{R}_2}$

- Case 1. $\boldsymbol{T}$ is an input time series, then $\hat{\varepsilon}_{Agg} = \sum_{i=1}^{k} r^{\varepsilon}_{T^i}$ where $r^{\varepsilon}_{T^i}$ is the reconstruction error in the error measures of T^i. [12]

- Case 2. $\boldsymbol{T}$ is a derived time series by applying the time series operators (recursively), Serialize(v, a, b), Shift($\boldsymbol{T}, k$), $\boldsymbol{T_1} + \boldsymbol{T_2}$, $\boldsymbol{T_1} - \boldsymbol{T_2}$ and $\boldsymbol{T_1} \times \boldsymbol{T_2}$. In this case, the aggregation operator $Agg = Sum(\boldsymbol{T})$ can be depicted as a tree. Figure 3.8 shows an example tree of the aggregation operator in the "correlation TSA" (Error measures in black color are precomputed offline during insertion time, while error measures in blue color are computed during the TSA processing time. The final error guarantees are in red color.). In order to compute $\hat{\varepsilon}_{Agg}$, we first calculate the error measures $\Phi(\boldsymbol{T}) = (\|\varepsilon_T\|_2, \|f_T\|_2, r^{\varepsilon}_T)$ for the root time series in the tree by propagating the error measures from the bottom time series to the root. Then we return the r^{ε}_T in the $\Phi(\boldsymbol{T})$ as the final error guarantee.

Next, we focus on computing the error measures for derived time series. We first explain the simpler case where each time series is a single segment. Table 3.9 shows the formulas for computing error measures for derived time series in this case. For the general scenario where

[12] Here we assume the aggregation operator aggregates the whole time series.

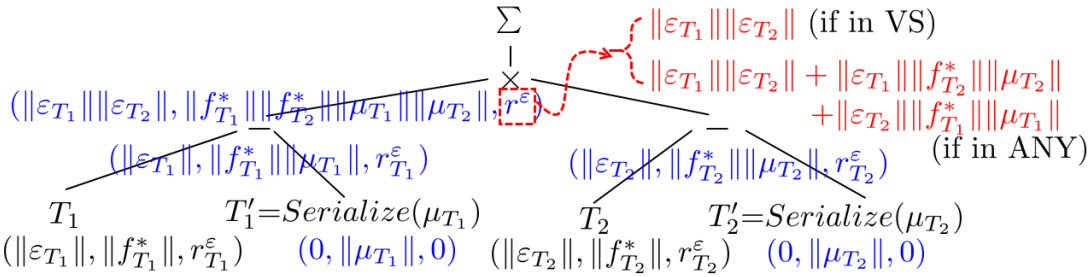


Figure 3.8. Example of error measures propagation.

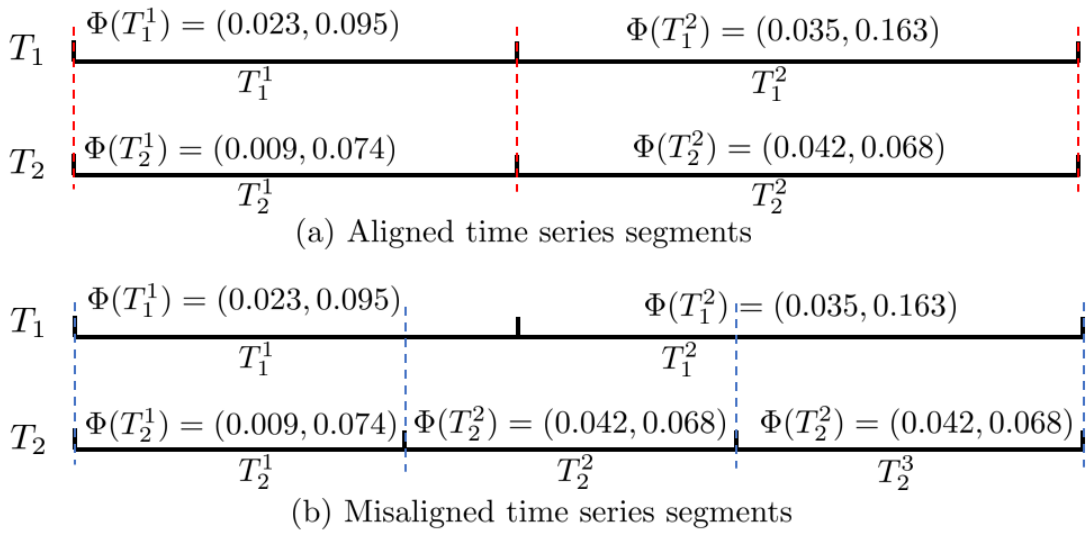


Figure 3.9. Example of aligned segments and misaligned segments.

multiple segments are involved in each input time series in the expression, there are two cases depending on whether the segments are aligned or not: If the i-th segment in T_1 has the same domain with the i-th segment in T_2 for all i, then T_1 and T_2 are *aligned*, otherwise, they are *misaligned*.

The computation of error guarantees of other expressions (i.e., Serialize(v, a, b), Shift$(\boldsymbol{T}, k)$, $\boldsymbol{T_1 + T_2}$ and $\boldsymbol{T_1 - T_2}$) is presented in Chapter 3.4.1.

3.4.1 Error Guarantees of Other Expressions

In this part, we present the error guarantees for the other core expressions, i.e., (i) Sum(Serialize(v, a, b)), (ii) Sum(Shift(T, k)), (iii) Sum$(T_1 + T_2)$, and (iv) Sum$(T_1 - T_2)$.

Table 3.9. Error measures propagation. $r^{\varepsilon}_{T=\boldsymbol{T_1}\times\boldsymbol{T_2}}$ has two possible computation methods. If the estimation function family forms a vector space, then we use the one in the grey background.

	Generated Error Measures		
	$\|\varepsilon_T\|_2$	$\|f_T\|_2$	r^{ε}_T
$\boldsymbol{T = T_1 + T_2}$	$\|\varepsilon_{T_1}\|_2 + \|\varepsilon_{T_2}\|_2$	$\|f_{T_1}\|_2 + \|f_{T_2}\|_2$	$r^{\varepsilon}_{T_1} + r^{\varepsilon}_{T_2}$
$\boldsymbol{T = T_1 - T_2}$	$\|\varepsilon_{T_1}\|_2 + \|\varepsilon_{T_2}\|_2$	$\|f_{T_1}\|_2 + \|f_{T_2}\|_2$	$r^{\varepsilon}_{T_1} + r^{\varepsilon}_{T_2}$
$\boldsymbol{T = T_1 \times T_2}$	$\|\varepsilon_{T_1}\|_2\|\varepsilon_{T_2}\|_2$ $+\|\varepsilon_{T_1}\|_2\|f_{T_2}\|_2$ $+\|\varepsilon_{T_2}\|_2\|f_{T_2}\|_2$	$\|f_{T_1}\|_2\|f_{T_2}\|_2$	$\|\varepsilon_{T_1}\|_2\|\varepsilon_{T_2}\|_2$ $+\|\varepsilon_{T_1}\|_2\|f_{T_2}\|_2$ $+\|\varepsilon_{T_2}\|_2\|f_{T_2}\|_2$

Error guarantee of Sum(Serialize(v, a, b)). For the time series T=Serialize(v, a, b), the estimation function is $f^*_T = v$ [13], then the error measures stored by Plato are ($\|\varepsilon_T\|_2 = 0$, $\|f_T\|_2 = v\sqrt{b-a+1}$, $r^{\varepsilon}_T = 0$). The error guarantee of Sum(Serialize(v, a, b)) is $r^{\varepsilon}_T = 0$.

Error guarantee of Sum(Shift(T, k)). For the time series T=Shift(T, k), we need to use the error measures ($\|\varepsilon_T\|_2$, $\|f_T\|_2$, r^{ε}_T) defined in domain $[a+k, b+k]$. Then the error guarantee of Sum(Shift(T, k)) is r^{ε}_T.

Error guarantees of Sum($T_1 + T_2$) and Sum($T_1 - T_2$). Given two time series $T_1 = (T^1_1, ..., T^{k_1}_1)$ and $T_2 = (T^1_2, ..., T^{k_2}_2)$. Then the error measures of $T = T_1 + T_2$ are ($\|\varepsilon_T\|_2$, $\|f_T\|_2$, r^{ε}_T) where $\|\varepsilon_T\|_2 = \sum_i^{k_1} \|\varepsilon_{T^i_1}\|_2 + \sum_i^{k_2} \|\varepsilon_{T^i_2}\|_2$, $\|f_T\|_2 = \sum_i^{k_1} \|f_{T^i_1}\|_2 + \sum_i^{k_2} \|f_{T^i_2}\|_2$, and $r^{\varepsilon}_T = \sum_i^{k_1} r^{\varepsilon}_{T^i_1} + \sum_i^{k_2} r^{\varepsilon}_{T^i_2}$. And the error guarantees of Sum$(T_1 + T_2)$ is $r^{\varepsilon}_T = \sum_i^{k_1} r^{\varepsilon}_{T^i_1} + \sum_i^{k_2} r^{\varepsilon}_{T^i_2}$. The error measures of $T_1 - T_2$ are the same with those of $T_1 + T_2$.

In the following, we will show how to compute the most challenging error guarantee $\hat{\varepsilon}_{Sum(T_1 \times T_2)}$ in both aligned and misaligned cases in Chapter 3.4.2 and Chapter 3.4.3 respectively.

3.4.2 Error Guarantee on Aligned Segments

Notations. Given a time series $\boldsymbol{T} = (\boldsymbol{T}[a], ..., \boldsymbol{T}[b])$ and the estimation function f^*_T of T, $\boldsymbol{\varepsilon}_T = \boldsymbol{T} - f^*_T = (\boldsymbol{T}[a] - f^*_T(a), ..., \boldsymbol{T}[b] - f^*_T(b))$ is the *vector of errors* produced by the estimation

[13] Under the reasonable assumption that any practical family will also include the constant function.

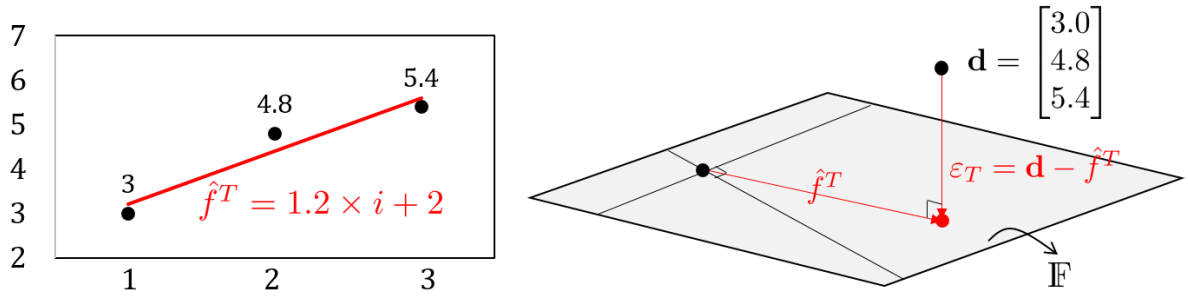
function. In the following, $\boldsymbol{T}$, f_T^* and $\boldsymbol{\varepsilon}$ are all regarded as vectors. $\langle f_1, f_2\rangle = \sum_{i=a}^{b} f_1(i)f_2(i)$ is the inner product of f_1 and f_2. $V|_{[a,b]}$ is a *restriction* operation, which restricts a vector V to the domain [a,b]. Recall a time series segment is a subsequence of a time series. Thus, a segment is the *restriction* of a time series $\boldsymbol{T}$ from a bigger domain $[a, b]$ into a smaller domain $[a', b'] \subseteq [a, b]$, denoted as $\boldsymbol{T}|_{[a',b']}$. Figure 3.7(b) visualizes the restriction operator. For example, consider a time series $\boldsymbol{T} = (1, 4, [1.2, 1.3, 1.3, 1.2])$, then $\boldsymbol{T}|_{[2,3]} = (2, 3, [1.3, 1.3])$ is a restriction of $\boldsymbol{T}$. Note that $\boldsymbol{T}|_{[a',b']}[i] = T[i]$ for all $i \in [a', b']$.

Given two compressed time series representation $L_{T_1} = (\tilde{T_1^1}, ..., \tilde{T_1^k})$ and $L_{T_1} = (\tilde{T_2^1}, ..., \tilde{T_2^k})$ for the *aligned* time series $T_1 = (T_1^1, ..., T_1^k)$ and $T_2 = (T_2^i, ..., T_2^k)$ where $T_1^i = T_1|_{[a_i,b_i]}$ and $T_2^i = T_2|_{[a_i,b_i]}$. Notice T_1^i and T_2^i have the same domain, i.e., $[a_i, b_i]$, for all $i \in [1, k]$. For any estimation function family, the error guarantee of $\mathsf{Sum}(\boldsymbol{T_1} \times \boldsymbol{T_2})$ on aligned time series is:

$$
\begin{aligned}
\varepsilon &= \Big| \sum_{i=a}^{b} \boldsymbol{T}_1[i]\boldsymbol{T}_2[i] - \sum_{i=a}^{b} f^*_{T_1}(i) f^*_{T_2}(i) \Big| \\
&= \Big| \sum_{i=1}^{k} \Big(\sum_{j=a_i}^{b_i} \boldsymbol{T}_1[i]\boldsymbol{T}_2[i] - \sum_{j=a_i}^{b_i} f^*_{T_1}(i) f^*_{T_2}(i) \Big) \Big| \\
&= \Big| \sum_{i=1}^{k} \Big(\langle \boldsymbol{\varepsilon}_{T_1^i}, f^*_{T_2^i} \rangle + \langle \boldsymbol{\varepsilon}_{T_2^i}, f^*_{T_1^i} \rangle + \langle \boldsymbol{\varepsilon}_{T_1^i}, \boldsymbol{\varepsilon}_{T_2^i} \rangle \Big) \Big| \\
&\leq \Big| \sum_{i=1}^{k} \langle \boldsymbol{\varepsilon}_{T_1^i}, f^*_{T_2^i} \rangle \Big| + \Big| \sum_{i=1}^{k} \langle \boldsymbol{\varepsilon}_{T_2^i}, f^*_{T_1^i} \rangle \Big| + \Big| \sum_{i=1}^{k} \langle \boldsymbol{\varepsilon}_{T_1^i}, \boldsymbol{\varepsilon}_{T_2^i} \rangle \Big| \\
&\leq \sum_{i=1}^{k} \Big(\|\varepsilon_{T_1^i}\|_2 \|\varepsilon_{T_2^i}\|_2 + \|\varepsilon_{T_1^i}\|_2 \|f_{T_2^i}\|_2 + \|f_{T_1^i}\|_2 \|\varepsilon_{T_2^i}\|_2 \Big)
\end{aligned}
\tag{3.2}
$$

The last inequality is obtained by Applying the Hölder inequality [23].

Example 7. *Consider the two aligned time series in Figure 3.9(a). Both T_1 and T_2 are partitioned into two segments in this case, i.e., (T_1^1, T_1^2) and (T_2^1, T_2^2). Plato stores the error measures $\Phi(T_i^j)$ for each segment T_i^j. For instance, $\Phi(T_1^1) = (\|\varepsilon_{T_1^1}\|_2, \|f_{T_1^1}\|_2, r^{\varepsilon}_{T_1^1}) = (0.023, 0.95, 0)$. Then the error guarantee of $\mathsf{Sum}(\boldsymbol{T_1} \times \boldsymbol{T_2})$ on T_1 and T_2 is computed as $(\|\varepsilon_{T_1^1}\|_2\|\varepsilon_{T_2^1}\|_2 + \|\varepsilon_{T_1^1}\|_2\|f_{T_2^1}\|_2 + \|f_{T_1^1}\|_2\|\varepsilon_{T_2^1}\|_2) + (\|\varepsilon_{T_1^2}\|_2\|\varepsilon_{T_2^2}\|_2 + \|\varepsilon_{T_1^2}\|_2\|f_{T_2^2}\|_2 + \|f_{T_1^2}\|_2\|\varepsilon_{T_2^2}\|_2) = (0.023 \times 0.009 + 0.023 \times$*



(a) Example estimation function (b) Visualization of orthogonal property

Figure 3.10. Example of orthogonal projection. (a) shows the estimation function for three data points. (b) visualizes the orthogonal projection of the three data points onto the 2-dimensional plane $\mathbb{F}$.

$0.074 + 0.095 \times 0.009) + (0.035 \times 0.042 + 0.035 \times 0.068 + 0.163 \times 0.042) = 0.01346$.

3.4.2.1 Orthogonal projection optimization

If the estimation function family forms a vector space (VS), [14] then we can apply the *orthogonal projection property* in VS to significantly reduce the error guarantee of $sum(T_1 \times T_2)$ from Formula 3.2 to Formula 3.3.

$$\varepsilon = \Big| \sum_{i=1}^{k} \Big(\underbrace{\langle \boldsymbol{\varepsilon}_{T_1^i}, f^*_{T_2^i} \rangle}_{=0 \ in \ VS} + \underbrace{\langle \boldsymbol{\varepsilon}_{T_2^i}, f^*_{T_1^i} \rangle}_{=0 \ in \ VS} + \langle \boldsymbol{\varepsilon}_{T_1^i}, \boldsymbol{\varepsilon}_{T_2^i} \rangle \Big) \Big|$$
$$\leq \sum_{i=1}^{k} \Big(\|\varepsilon_{T_1^i}\|_2 \|\varepsilon_{T_2^i}\|_2 \Big) \tag{3.3}$$

Example 8. *Consider the two aligned time series in Figure 3.9(a) again. The estimation function family is polynomial function family, it is VS. Based on Formula 3, the error guarantee for* $Sum(\boldsymbol{T_1} \times \boldsymbol{T_2})$ *is* $\|\varepsilon_{T_1^1}\|_2 \times \|\varepsilon_{T_2^1}\|_2 + \|\varepsilon_{T_1^2}\|_2 \times \|\varepsilon_{T_2^2}\|_2 = 0.023 \times 0.009 + 0.035 \times 0.042 = 0.001677$. *This error guarantee is about* $8\times$ *smaller than that in Example 7 (i.e.,* 0.01346*), where we did not take into account that the function family is VS.*

Orthogonal projection property. Example 8 indicates the power of the orthogonal

[14] A vector space is a set that is closed under finite vector addition and scalar multiplication. http://mathworld.wolfram.com/VectorSpace.html.

projection optimization. Lemma 9 is a proof of Formula 3.3.

Lemma 9. *(Orthogonal Projection Property) Let* $\mathbb{F}$ *be a function family forms a vector space* VS *and* $f_T^* \in \mathbb{F}$ *be the estimation function of time series* T*. Then* f_T^* *is the orthogonal projection of* $\mathbf{T}$ *onto* $\mathbb{F}$ *[72].*

Lemma 9 implies that $\boldsymbol{\varepsilon}_T$ is orthogonal to any function $f_T \in \mathbb{F}$, which means $\langle \boldsymbol{\varepsilon_T}, \boldsymbol{f_T} \rangle = 0$. Therefore, given any two aligned segments T_1^i and T_2^i, as both $f_{T_1^i}^*$ and $f_{T_2^i}^*$ are in VS, thus $\langle \boldsymbol{\varepsilon}_{T_1^i}, f_{T_2^i}^* \rangle = 0$ and $\langle \boldsymbol{\varepsilon}_{T_2^i}, f_{T_1^i}^* \rangle = 0$.

For visualization purposes, consider a time series with three data points $T = (1, 3, [3.0, 4.8, 5.4])$ and let $\mathbb{F}$ be the 1-degree polynomial function family (i.e., 2-dimensional). The estimation function that minimizes the error to the original data is $f_T^* = 1.2 \times i + 2$ (Figure 3.10(a)). As shown in Figure 3.10(b), f_T^* is the orthogonal projection of $\boldsymbol{T}$ onto $\mathbb{F}$. The error vector is $\boldsymbol{\varepsilon}_T = (-0.2, 0.4, -0.2)$. Based on Lemma 9, for any candidate estimation function $f = \alpha \times i + \beta$ $(\alpha, \beta \in R)$, we have $\langle \boldsymbol{\varepsilon_T}, \boldsymbol{f} \rangle = 0.8\alpha - 0.8\alpha + 0.4\beta - 0.4\beta = 0$.

Elimination of r_T^ε. We can get an extra benefit from the orthogonal projection property in saving space, i.e., the error measure r_T^ε can be avoided as it is guaranteed to be 0. This is because $r_T^\varepsilon = \langle T - f_T^*, 1 \rangle$ and 1 is a constant function in the function family in VS. According to Lemma 9, we know $\langle T - f_T^*, 1 \rangle = 0$. Therefore, we have $r_T^\varepsilon = 0$.

Amplitude-independent (AI). The orthogonal projection optimization can significantly reduce the error guarantees. It allows the error guarantees to get rid of the *amplitudes* of the original time series values (referring to $\|f_T\|_2$) by only consider the reconstruction error (referring to $\|\varepsilon_T\|_2$) of each time series. The error guarantees provided by Plato in VS are called *amplitude-independent (AI)* error guarantees.

3.4.3 Error Guarantee on Misaligned Segments

Given two compressed time series representation $L_{T_1} = (\tilde{T_1^1}, ..., \tilde{T_1^{k_1}})$ and $L_{T_1} = (\tilde{T_2^1}, ..., \tilde{T_2^{k_2}})$ for the misaligned time series $T_1 = (T_1^1, ..., T_1^{k_1})$ and $T_2 = (T_2^1, ..., T_2^{k_2})$ where the

domains of T_1^i and T_2^i are $[a_i^1, b_i^1]$ and $[a_i^2, b_i^2]$ respectively. The major challenge in the misaligned case is that for a domain $[a_1^i, b_1^i]$, the error measures of the segment $T_1|_{[a_1^i, b_1^i]}$ are precomputed, however, the error measures of the segment $T_2|_{[a_1^i, b_1^i]}$ may be unknown as $T_2|_{[a_1^i, b_1^i]}$ in general is not one of the segments $T_2^1, ..., T_2^{k_2}$.

Let $\Pi_{T,[a,b]}$ be the set of segments in T covering the domain $[a, b]$. For example, consider the two misaligned time series T_1 and T_2 in Figure 3.9(b), $\Pi_{T_2,[a_1^1,b_1^1]} = \{T_2^1, T_2^2\}$ as the segments T_2^1 and T_2^2 in T_2 cover the domain $[a_1^1, b_1^1]$. [15] If any kinds of function families are allowed, i.e., in ANY, the error guarantee $\hat{\varepsilon}$ of $\mathsf{Sum}(\boldsymbol{T_1} \times \boldsymbol{T_2})$ on misaligned time series is:

$$
\begin{aligned}
\hat{\varepsilon} &= \left| \sum_{i=a}^{b} \boldsymbol{T}_1[i]\boldsymbol{T}_2[i] - \sum_{i=a}^{b} f^*_{T_1}(i) f^*_{T_2}(i) \right| \\
&\leq \left| \langle \boldsymbol{\varepsilon_{T_1}}, f^*_{T_2} \rangle \right| + \left| \langle \boldsymbol{\varepsilon_{T_2}}, f^*_{T_1} \rangle \right| + \left| \langle \boldsymbol{\varepsilon_{T_1}}, \boldsymbol{\varepsilon_{T_2}} \rangle \right| \\
&= \left| \sum_{i=1}^{k_1} \langle \boldsymbol{\varepsilon_{T_1^i}}, f^*_{T_2}|_{[a_1^i, b_1^i]} \rangle \right| + \left| \sum_{i=1}^{k_2} \langle \boldsymbol{\varepsilon_{T_2^i}}, f^*_{T_1}|_{[a_2^i, b_2^i]} \rangle \right| + \left| \langle \boldsymbol{\varepsilon_{T_1}}, \boldsymbol{\varepsilon_{T_2}} \rangle \right| \\
&\leq \sum_{i=1}^{k_1} \left(\|\varepsilon_{T_1^i}\|_2 \Big(\sum_{j \in \Pi_{T_2,[a_1^i,b_1^i]}} \|f_{T_2^j}\|_2^2 \Big)^{\frac{1}{2}} \right) + \sum_{i=1}^{k_2} \left(\|\varepsilon_{T_2^i}\|_2 \Big(\sum_{j \in \Pi_{T_1,[a_2^i,b_2^i]}} \|f_{T_1^j}\|_2^2 \Big)^{\frac{1}{2}} \right) \quad (1) \\
&\quad + \left| \langle \boldsymbol{\varepsilon_{T_1}}, \boldsymbol{\varepsilon_{T_2}} \rangle \right| \quad (2)
\end{aligned} \tag{3.4}
$$

Formula 3.4 is a stepping stone towards producing the final formula as the computation of $|\langle \boldsymbol{\varepsilon_{T_1}}, \boldsymbol{\varepsilon_{T_2}} \rangle|$ (Formula 3.4②) has not been given yet. It will be discussed in Chapter 3.4.3.1. Chapter 3.4.3.2 discusses how to apply the orthogonal property optimization to improve Formula 3.4①.

3.4.3.1 Segment combination selection

To compute $|\langle \boldsymbol{\varepsilon_{T_1}}, \boldsymbol{\varepsilon_{T_2}} \rangle|$, one straightforward method (called IS) is to use the domains of segments in T_1 and T_2 independently, then choose the one with minimal value. Let's first see

[15] If time series T_1 and T_2 are aligned, then $\Pi_{T_2,[a_1^i,b_1^i]}$ always returns one single segment.

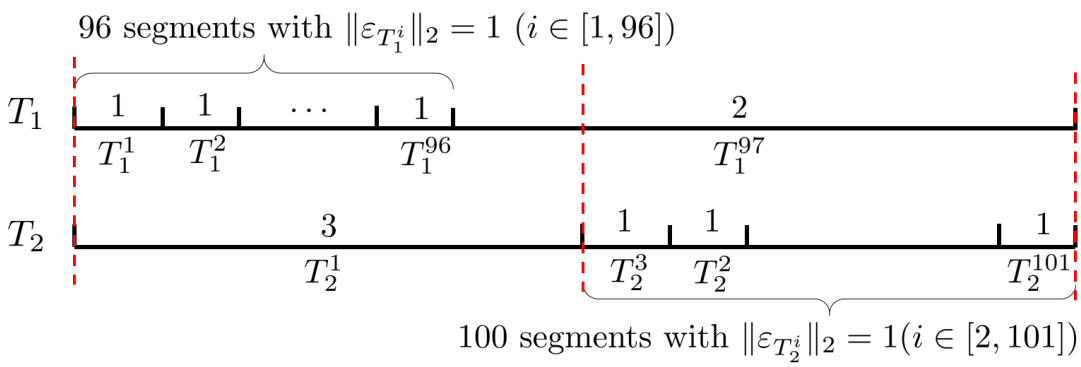


Figure 3.11. Example of segment combination selection.

how to compute $|\langle \boldsymbol{\varepsilon_{T_1}}, \boldsymbol{\varepsilon_{T_2}} \rangle|$ with the domains of segments in T_1.

$$|\langle \boldsymbol{\varepsilon_{T_1}}, \boldsymbol{\varepsilon_{T_2}} \rangle| \leq \sum_{i=1}^{k_1} \left| \langle \boldsymbol{\varepsilon_{T_1}}|_{[\boldsymbol{a_1^i}, \boldsymbol{b_1^i}]}, \boldsymbol{\varepsilon_{T_2}}|_{[\boldsymbol{a_1^i}, \boldsymbol{b_1^i}]} \rangle \right| = \sum_{i=1}^{k_1} \left| \langle \boldsymbol{\varepsilon_{T_1^i}}, \boldsymbol{\varepsilon_{T_2}}|_{[\boldsymbol{a_1^i}, \boldsymbol{b_1^i}]} \rangle \right|$$
$$\leq \sum_{i=1}^{k_1} \left(\|\varepsilon_{T_1^i}\|_2 \left(\sum_{j \in \Pi_{T_2, [a_1^i, b_1^i]}} \|\varepsilon_{T_2^j}\|_2^2 \right)^{\frac{1}{2}} \right)$$

In the last step of the above Formula, $T_2|_{[a_1^i, b_1^i]}$ is not a segment that Plato precomputed in T_2. Thus, we need to use all the segments in T_2 covering $[a_1^i, b_1^i]$, i.e., $\Pi_{T_2, [a_1^i, b_1^i]}$. Similarly, we can compute $|\langle \boldsymbol{\varepsilon_{T_1}}, \boldsymbol{\varepsilon_{T_2}} \rangle|$ according to the domains of segments in T_2. Finally, IS chooses the minimal one between them. That is, the output of IS is:

$$\min \Bigg(\sum_{i=1}^{k_1} \left(\|\varepsilon_{T_1^i}\|_2 \left(\sum_{j \in \Pi_{T_2, [a_1^i, b_1^i]}} \|\varepsilon_{T_2^j}\|_2^2 \right)^{\frac{1}{2}} \right), \sum_{i=1}^{k_2} \left(\|\varepsilon_{T_2^i}\|_2 \left(\sum_{j \in \Pi_{T_1, [a_2^i, b_2^i]}} \|\varepsilon_{T_1^j}\|_2^2 \right)^{\frac{1}{2}} \right) \Bigg)$$

However, IS does not produce tight guarantees, Plato does not use it. Next, we show the tight computation called *OS*, which is used by Plato.

Optimal strategy (OS). Algorithm OS first computes an error distribution array E_{T_1}

(resp. E_{T_2}) for T_1 (resp. T_2) (line 2) according to the domains of the segments as follows:

$$E_{T_1} = \Big\{ \|\varepsilon_{T_1^i}\|_2 \times \Big(\sum_{j \in \Pi_{T_2,[a_1^i,b_1^i]}} \|\varepsilon_{T_2^i}\|_2^2 \Big)^{\frac{1}{2}} \Big| 1 \leq i \leq k_1 \Big\}$$

$$E_{T_2} = \Big\{ \|\varepsilon_{T_2^i}\|_2 \times \Big(\sum_{j \in \Pi_{T_1,[a_2^i,b_2^i]}} \|\varepsilon_{T_1^i}\|_2^2 \Big)^{\frac{1}{2}} \Big| 1 \leq i \leq k_2 \Big\}$$

Then OS increases ε_1 (resp. ε_2) by adding the values from E_{T_1} (resp. E_{T_2}) (lines 4-7) and checks whether the current domain achieves the minimal errors (lines 8-17). If yes, OS adds the current domain (either $[start, b_1^{i_1}]$ or $[start, b_2^{i_2}]$) to the final segment combination list. After that, OS starts from a new domain and repeats the previous steps until all the segments are processed. The time complexity of OS is $O(k_1 + k_2)$.

Let $OPT(L_{T_1}, L_{T_2})$ be the segment combination returned by OS. Then $|\langle \boldsymbol{\varepsilon_{T_1}}, \boldsymbol{\varepsilon_{T_2}} \rangle|$ is computed as follows:

$$\begin{aligned} |\langle \boldsymbol{\varepsilon_{T_1}}, \boldsymbol{\varepsilon_{T_2}} \rangle| &\leq \sum_{[a,b] \in OPT(L_{T_1}, L_{T_2})} \Big| \langle \boldsymbol{\varepsilon_{T_1}}|_{[a,b]}, \boldsymbol{\varepsilon_{T_2}}|_{[a,b]} \rangle \Big| \\ &\leq \sum_{[a,b] \in OPT(L_{T_1}, L_{T_2})} \Big(\Big(\sum_{i \in \Pi_{T_1,[a,b]}} \|\boldsymbol{\varepsilon_{T_1^i}}\|_2^2 \Big)^{\frac{1}{2}} \Big(\sum_{i \in \Pi_{T_2,[a,b]}} \|\boldsymbol{\varepsilon_{T_2^i}}\|_2^2 \Big)^{\frac{1}{2}} \Big) \end{aligned}$$

OS provides the optimal segment combination that produces the minimum $|\langle \boldsymbol{\varepsilon_{T_1}}, \boldsymbol{\varepsilon_{T_2}} \rangle|$. The tightness proof is presented as follows.

Proof. We use a proof by induction to show that the error guarantee produced by the segment combination returned by OS (Algorithm) is optimal.

Let $OPT(\tilde{T_1}, \tilde{T_2}) = \{[a_i, b_i] | i \in [1, m]\}$ be the segment combination returned by OS. First, let's see the base case where $OPT(\tilde{T_1}, \tilde{T_2}) = \{[a_1, b_1]\}$ has only one domain. There are two cases depending on $b_1 = b_1^1$ or $b_1 = b_2^t$ where $\Pi_{T_2,[a_1,b_1]} = \{T_2^1, ..., T_2^t\}$.

Case 1: $b_1 = b_1^1$. Since OS chooses $[a_1, b_1^1]$ as the domain, then $b_2^1 \leq b_1^1$. Otherwise, OS does not choose $[a_1, b_1^1]$. This is because, (i) if $E_{T_1}[0] \geq E_{T_2}[0]$ then OS will choose $[a_1, b_2^1]$

instead; or (ii) if $E_{T_1}[0] < E_{T_2}[0]$, then OS can not enter the loop in lines 8 - 17. Since $b_2^1 \leq b_1^1$, then we know $E_{T_1}[0] \leq E_{T_2}[0]$, so the error guarantee is $\|\varepsilon_{T_1^1}\|_2 \|\varepsilon_{T_2^1}\|_2$, which is the minimal error guarantee in domain $[a_1, b_1]$. Assume we split the domain $[a_1, b_1]$ into p ($p \geq 2$) sub-domains $[a_1, c_1], [c_1, c_2], ..., [c_{p-1}, b_1]$, then the error guarantee is $p\|\varepsilon_{T_1^1}\|_2 \|\varepsilon_{T_2^1}\|_2$, therefore, domain $[a_1, b_1] = [a_1^1, b_1^1]$ produces the minimal error guarantee.

Case 2: $b_1 = b_2^t$. Since OS chooses $[a_1, b_2^t]$ as the domain, we know that the error guarantee is

$$\Big(\sum_{i \in \Pi_{T_2, [a_2^1, b_2^t]}} \|\varepsilon_{T_2^1}\|_2^2 \Big)^{\frac{1}{2}} \Big(\sum_{i \in \Pi_{T_1, [a_2^1, b_2^t]}} \|\varepsilon_{T_1^1}\|_2^2 \Big)^{\frac{1}{2}}$$

which is less than $\|\varepsilon_{T_1^1}\|_2 (\sum_{i \in \Pi_{T_2, [a_1^1, b_1^1]}} \|\varepsilon_{T_2^1}\|_2^2)^{\frac{1}{2}}$. If we split $[a_2^1, b_2^t]$ into several sub-domains, the error guarantee is greater than $\|\varepsilon_{T_1^1}\|_2 (\sum_{i \in \Pi_{T_2, [a_1^1, b_1^1]}} \|\varepsilon_{T_2^1}\|_2^2)^{\frac{1}{2}}$. Thus, $[a_1, b_1] = [a_2^1, b_2^t]$ produces the minimal error guarantee.

Suppose $OPT(\tilde{T_1}, \tilde{T_2}) = \{[a_i, b_i] | i \in [1, m-1]\}$ produces the minimal error guarantee, then for the case $OPT(\tilde{T_1}, \tilde{T_2}) = \{[a_i, b_i] | i \in [1, m]\}$, we only need to prove the last domain $[a_m, b_m]$ produces the minimal error guarantee, which is the same to the base case. □

Example 10. *Consider the two misaligned time series in Figure 3.11. The value of* $\|\varepsilon_{T_i^j}\|_2$ *for each segment* T_i^j *is labeled there. OS produces the segment combination* $S = \{[a_1^1, b_2^1], [b_2^1, b_2^{101}]\}$ *as visualized by the red lines. Then* $|\langle \boldsymbol{\varepsilon_{T_1}}, \boldsymbol{\varepsilon_{T_2}} \rangle| = (3 \times (96 \times 1^2 + 2^2)^{\frac{1}{2}}) + (2 \times (100 \times 1^2)^{\frac{1}{2}}) = 3 \times 10 + 2 \times 10 = 50$. *However, IS outputs* $|\langle \boldsymbol{\varepsilon_{T_1}}, \boldsymbol{\varepsilon_{T_2}} \rangle| = min((3 \times 96 + 2 \times \sqrt{100 + 9}), (3 \times \sqrt{96 + 2^2} + 100 \times 2)) = \min(308.88, 230) = 230$, *which is* $4.6\times$ *larger than the result returned by OS.*

3.4.3.2 Orthogonal projection optimization

In this part, we present how to apply orthogonal property optimization to improve Formula 3.4①. Recall that in the aligned case (if the function family is in VS) we can apply the

Algorithm: Optimal segment combination (OS)

Input: Compressed segment representations L_{T_1}, L_{T_2}
Output: A segment combination OPT

$\varepsilon_1 = 0, \varepsilon_2 = 0, i_i = 0, i_2 = 0, start = 0, OPT = \emptyset, current = \emptyset$;
Compute E_{T_1} and E_{T_2};
while $i_1 < k_1$ *or* $i_2 < k_2$ **do**
    if $b_1^{i_1} \leq b_2^{i_2}$ **then**
        $\varepsilon_1 += E_{T_1}[i_1 ++]$;
    else
        $\varepsilon_2 += E_{T_2}[i_2 ++]$;
    if $\varepsilon_1 \leq \varepsilon_2$ *AND* $b_1^{i_1} \geq b_2^{i_2}$ **then**
        $current = [start, b_1^{i_1}]$;
        $OPT \leftarrow OPT \cup \{current\}$;
        $start = b_1^{i_1} + 1$;
        $\varepsilon_2 \leftarrow \varepsilon_1$;
    if $\varepsilon_2 \leq \varepsilon_1$ *AND* $b_2^{i_2} \geq b_1^{i_1}$ **then**
        $current = [start, b_2^{i_2}]$;
        $OPT \leftarrow OPT \cup \{current\}$;
        $start = b_2^{i_2} + 1$;
        $\varepsilon_1 \leftarrow \varepsilon_2$;
Return OPT;

orthogonal property optimization to guarantee $\langle \varepsilon_{T_1^i}, f^*_{T_2}|_{[a_1^i, b_1^i]} \rangle = 0$. This is because $f^*_{T_2}|_{[a_1^i, b_1^i]} = f^*_{T_2^i}$, which is a function in the family. However, in misaligned case $\langle \varepsilon_{T_1^i}, f^*_{T_2}|_{[a_1^i, b_1^i]} \rangle$ cannot be guaranteed to be 0 since $f^*_{T_2}|_{[a_1^i, b_1^i]}$ may not be a function in the family. For example, in Figure 3.11 $T_2|_{[a_2^1, b_2^1]}$ is not a pre-computed segment in T_2, it is just a subsegment. The restriction of the estimation function $f^*_{T_2^1}$ to this sub-domain $f^*_{T_2}|_{[a_1^1, b_1^1]}$ may not be a function in the family anymore.

To guarantee the restriction of the function from a bigger domain to a smaller domain is still in the same function family, we identify a function family group called linear scalable function family (LSF), which is subset of VS but superset of the polynomial function family.

Linear Scalable Function Family (LSF). Informally, a linear scalable family is a function family such that for any function f in that family and any translation $a - a'$, there is a function f' in that family such that $f'(x + a - a') = f(x)$ for all x in the domain. Definition 11

gives the formal definition.

Definition 11 (Linear scalable family (LSF))**.** *Let* $\mathbb{F}$ *be a function family defined in domain* $[a, b]$*,* $\mathbb{F}$ *is a linear scalable family if for any function* $f \in \mathbb{F}$ *and any range* $[a', b'] \subseteq [a, b]$*, there exists a function* $f' \in \mathbb{F}$ *such that* $\mathsf{Shift}(f|_{[a',b']}, a - a') = f'|_{[a,a+b'-a']}$.

Lemma 12. *The polynomial family belongs to the linear scalable family.*

Proof. Let $\mathbb{F} = \{\sum_i \alpha_i t^i | \alpha_i \in R\}$ be a polynomial function family defined on $[a, b]$. The restriction of $f \in \mathbb{F}$ on $[a', b'] \subseteq [a, b]$ is $f|_{[a',b']} = (a', b', [\sum_i \alpha_i (a')^i, ..., \sum_i \alpha_i (b')^i])$. The shift of $f|_{[a',b']}$ to $a - a'$ steps is $\mathsf{Shift}(f|_{[a',b']}, a - a') = (a, a + b' - a', [\sum_i \alpha_i (a')^i, ..., \sum_i \alpha_i (b')^i])$. $[\sum_i \alpha_i (a')^i, ..., \sum_i \alpha_i (b')^i]$ can be transformed into $[\sum_i \beta_i (a)^i, ..., \sum_i \beta_i (a + b' - a')^i]$ such that $\beta_i = \frac{\alpha_i (a'+k)^i}{(a+k)^k}$ for all $i \in [a, a + b' - a']$. Let $f' = \sum_i \beta_i t^i$ be a function in $\mathbb{F}$. Thus $f'|_{[a,a+b'-a']} = [\sum_i \beta_i (a)^i, ..., \sum_i \beta_i (a + b' - a')^i] = Shift(f|_{[a',b']}, a - a')$. □

Recall that, in this paper, we study three different function family groups, i.e., ANY, VS, and LSF. Figure 3.5 shows the relation of the three function family groups and also provides example function families for each group.

In the following, we present how to use the orthogonal projection optimization in the misaligned case to improve Formula 4①. Let $\boldsymbol{f_{T_1}}$ (resp. $\boldsymbol{f_{T_2}}$) be the function created from the concatenation of the individual estimation functions on the segments T_1^i ($i \in [1, k_1)$ (resp. T_2^j ($j \in [1, k_2]$)). That is $\boldsymbol{f_{T_1}}|_{[\boldsymbol{a_1^i}, \boldsymbol{b_1^i}]} = f^*_{T_1^i}$ for all $i \in [1, k_1]$ and $\boldsymbol{f_{T_2}}|_{[\boldsymbol{a_2^i}, \boldsymbol{b_2^i}]} = f^*_{T_2^i}$ for all $i \in [1, k_2]$. Then the Equation 4① in the misaligned environment can be reduced as follows. We highlight the parts that would disappear if the segments were aligned.

$$\sum_{i=1}^{k_1} \left(\|\varepsilon_{T_1^i}\|_2 \times \overbrace{\|\boldsymbol{f_{T_2}}|_{[\boldsymbol{a_1^i}, \boldsymbol{b_1^i}]} - \boldsymbol{f^*_{T_1^i}}\|_2}^{=0 \ if \ aligned} \right)$$
$$+ \sum_{i=1}^{k_2} \left(\|\varepsilon_{T_2^i}\|_2 \times \overbrace{\|\boldsymbol{f_{T_1}}|_{[\boldsymbol{a_2^i}, \boldsymbol{b_2^i}]} - \boldsymbol{f^*_{T_2^i}}\|_2}^{=0 \ if \ aligned} \right) \qquad (5)$$

The proof of the tightness is as follows.

Proof. Let $\varepsilon_{Sum(T_1 \times T_2)}$ be the true error of $Sum(T_1 \times T_2)$.

$$\begin{aligned}\varepsilon_{Sum(T_1 \times T_2)} &= |\langle \boldsymbol{\varepsilon}_{T_1}, \boldsymbol{f}_{T_2} \rangle + \langle \boldsymbol{\varepsilon}_{T_2}, \boldsymbol{f}_{T_1} \rangle + \langle \boldsymbol{\varepsilon}_{T_1}, \boldsymbol{\varepsilon}_{T_2} \rangle| \\ &\leq |\langle \boldsymbol{\varepsilon}_{T_1}, \boldsymbol{f}_{T_2} \rangle| + |\langle \boldsymbol{\varepsilon}_{T_2}, \boldsymbol{f}_{T_1} \rangle| + |\langle \boldsymbol{\varepsilon}_{T_1}, \boldsymbol{\varepsilon}_{T_2} \rangle|\end{aligned}$$

The first term $|\langle \boldsymbol{\varepsilon}_{T_1}, \boldsymbol{f}_{T_2} \rangle|$ can be rewritten as

$$\begin{aligned}|\langle \boldsymbol{\varepsilon}_{T_1}, \boldsymbol{f}_{T_2} \rangle| &= \Big| \sum_{i=1}^{k_1} \langle \boldsymbol{\varepsilon}_{T_1}|_{[a_i,b_i]}, \boldsymbol{f}_{T_2}|_{[a_i,b_i]} \rangle \Big| \\ &= \Big| \sum_{i=1}^{k_1} \Big(\overbrace{\langle \boldsymbol{\varepsilon}_{T_1}|_{[a_1^i,b_1^i]}, f^*_{T_1^i} \rangle}^{=0} + \langle \boldsymbol{\varepsilon}_{T_1}|_{[a_1^i,b_1^i]}, \boldsymbol{f}_{T_2}|_{[a_1^i,b_1^i]} - f^*_{T_1^i} \rangle \Big) \Big| \\ &\leq \sum_{i=1}^{k_1} \Big| \Big(\langle \boldsymbol{\varepsilon}_{T_1}|_{[a_1^i,b_1^i]}, \boldsymbol{f}_{T_2}|_{[a_1^i,b_1^i]} - f^*_{T_1^i} \rangle \Big) \Big| \\ &\leq \sum_{i=1}^{k_1} \big\| \boldsymbol{\varepsilon}_{T_1}|_{[a_1^i,b_1^i]} \big\|_2 \big\| \boldsymbol{f}_{T_2}|_{[a_1^i,b_1^i]} - f^*_{T_1^i} \big\|_2 \\ &= \sum_{i=1}^{k_1} \|\varepsilon_{T_1^i}\|_2 \|\boldsymbol{f}_{T_2}|_{[a_1^i,b_1^i]} - f^*_{T_1^i}\|_2\end{aligned}$$

Similarly, we have:

$$|\langle \boldsymbol{\varepsilon}_{T_2}, \boldsymbol{f}_{T_1} \rangle| \leq \sum_{i=1}^{k_2} \Big(\|\varepsilon_{T_2^i}\|_2 \times \|\boldsymbol{f}_{T_1}|_{[a_2^i,b_2^i]} - f^*_{T_2^i}\|_2 \Big)$$

Recall that the computation of $|\langle \boldsymbol{\varepsilon}_{T_1}, \boldsymbol{\varepsilon}_{T_2} \rangle|$ is presented in Chapter 3.4.3.1. Combining the results of $|\langle \boldsymbol{\varepsilon}_{T_1}, \boldsymbol{\varepsilon}_{T_2} \rangle|$, $|\langle \boldsymbol{\varepsilon}_{T_1}, \boldsymbol{f}_{\boldsymbol{T_2}} \rangle|$, and $|\langle \boldsymbol{f}_{\boldsymbol{T_1}}, \boldsymbol{\varepsilon}_{T_2} \rangle|$ completes the proof. □

Efficient Computation of the Error Guarantee. Notice that both $\|\boldsymbol{f}_{\boldsymbol{T_2}}|_{[\boldsymbol{a_1^i},\boldsymbol{b_1^i}]} - \boldsymbol{f}^*_{\boldsymbol{T_1^i}}\|_2$ and $\|\boldsymbol{f}_{\boldsymbol{T_1}}|_{[\boldsymbol{a_2^i},\boldsymbol{b_2^i}]} - \boldsymbol{f}^*_{\boldsymbol{T_2^i}}\|_2$ can only be computed during query processing time, since only then the pairs of intersecting but misaligned segments become known. A brute force $O(n)$ method, where n is the size of the domain of the segment, would be to literally create the series of n data points predicted by the estimation functions and then perform the straightforward calcula-

tion/aggregation described by the formulas. Of course, such brute force approach would require CPU cycles that are proportional to conventional (non-approximate) query processing. We show that these formulas can be computed in $O(dim(\mathbb{F})^3)$ where $dim(\mathbb{F})$ is the dimension of the estimation function family. Obviously, the dimension is much smaller than the number of data points in a segment - that is why we employ compression in the first place. For example, for a 1-degree polynomial function family, $dim(\mathbb{F}) = 2$. The key intuition is to store the estimation function's coefficients in an orthonormal basis. The distance between two functions can be efficiently computed using the $dim(\mathbb{F})$ coefficients (in the orthonormal basis). Importantly, the orthonormal basis also allows us to compute the coefficients of the restriction of an estimation function in $O(dim(\mathbb{F})^3)$. The detailed algorithms and proofs complexity appear in the following.

Here we present how to compute $\|\boldsymbol{f_{T_2}}|_{[\boldsymbol{a_1^i},\boldsymbol{b_1^i}]} - \boldsymbol{f^*_{T_1^i}}\|_2$ and $\|\boldsymbol{f_{T_1}}|_{[\boldsymbol{a_2^i},\boldsymbol{b_2^i}]} - \boldsymbol{f^*_{T_2^i}}\|_2$ in $O(dim(\mathbb{F})^3)$. Let's first look into $\|\boldsymbol{f_{T_2}}|_{[\boldsymbol{a_1^i},\boldsymbol{b_1^i}]} - \boldsymbol{f^*_{T_1^i}}\|_2$.

$$
\begin{aligned}
&\|\boldsymbol{f_{T_2}}|_{[\boldsymbol{a_1^i},\boldsymbol{b_1^i}]} - \boldsymbol{f^*_{T_1^i}}\|_2 = \\
&\Big(\sum_{j \in \Pi_{T_2,[a_1^i,b_1^i]}} \|\boldsymbol{f_{T_2}}|_{[\boldsymbol{a_1^i},\boldsymbol{b_1^i}]\cap[\boldsymbol{a_2^j},\boldsymbol{b_2^j}]} - \boldsymbol{f^*_{T_1^i}}|_{[\boldsymbol{a_1^i},\boldsymbol{b_1^i}]\cap[\boldsymbol{a_2^j},\boldsymbol{b_2^j}]}\|_2^2 \Big)^{\frac{1}{2}} \\
&= \Big(\sum_{j \in \Pi_{T_2,[a_i,b_i]}} \|\Psi([a_2^i,b_2^i],[a_1^i,b_1^i]\cap[a_2^j,b_2^j]) f^*_{T_2^j} \\
&\quad - \Psi([a_1^i,b_1^i],[a_1^i,b_1^i]\cap[a_2^j,b_2^j]) f^*_{T_1^i}\|_2^2 \Big)^{\frac{1}{2}}
\end{aligned}
$$

where Ψ is an orthonormal basis transformation matrix, which can be computed in $O(dim(\mathbb{F})^3)$. $\Psi([a_2^i,b_2^i],[a_1^i,b_1^i]\cap[a_2^j,b_2^j])$ transforms the orthonormal basis from the domain $[a_2^i,b_2^i]$ to the sub-domain $[a_1^i,b_1^i]\cap[a_2^j,b_2^j]$. In the following, we will show the details of computing Ψ.

Given a function family $\mathbb{F}$, let $(\varphi_i^{[a,b]})_{1\le i\le dim(\mathbb{F})}$ be an orthonormal basis of $\mathbb{F}$ on the domain $[a,b]$ for the scalar product $\langle f_1,f_2\rangle = \sum_{i=a}^{b}(f_1(i)\times f_2(i))$ where $f_1,f_2\in\mathbb{F}$. Such orthonormal basis can be obtained by using the Gram$-$Schmidt process [37]. Given a domain $[a,b]$ and one sub-domain $[a',b']\subset[a,b]$, let $\Psi([a,b],[a',b'])$ be the basis transform matrix such

that

$$\Psi([a,b],[a',b'])_{i,j} = \langle \varphi_i^{[a,b]}|_{[a',b']}, \varphi_j^{[a',b']} \rangle$$

That is using Ψ, we can directly obtain the orthonormal basis for any sub-domain. The size of Ψ is $dim(\mathbb{F})^2$ and the computation of each $\Psi([a,b],[a',b'])_{i,j}$ is $O(dim(\mathbb{F}))$. Therefore, the overall cost of computing Ψ is $O(dim(\mathbb{F})^3)$.

Elimination of $\|f_T\|_2$. If T is compressed by a function in LSF [16], then $\|f_T\|_2$ can be safely eliminated. This is because the error guarantees provided by LSF can get rid of $\|f_T\|_2$ while those given by ANY or VS rely on $\|f_T\|_2$.

3.5 Error Guarantee Computation on Segment Trees

The previous chapter studies the computation of error guarantees for TSAs over compressed segment lists. In this chapter, we first show how to compute error guarantees for anytime analytic queries with error budgets on compressed segment trees (Chapter 3.5.1). Then we propose an incremental error guarantee computation optimization (Chapter 3.5.2) to improve the performance. We focus on supporting queries with error budgets and it is straightforward to extend our algorithms to support queries with time budgets.

3.5.1 Error Guarantee Computation in Segment Tree

The main idea of supporting analytic queries with error budgets is to use a top-down navigation strategy. Accessing more nodes leads to smaller error guarantees. As shown in Algorithm EGC, it first gathers the currently accessed nodes $\mathscr{N}_1$ and $\mathscr{N}_2$ in the segment trees $\mathscr{T}_1$ and $\mathscr{T}_2$ (line 2). Then it computes the current error guarantee ε by using error measures stored in $\mathscr{N}_1$ and $\mathscr{N}_2$ [17] according to the formulas provided in Chapter 3.4 (line 3) . If ε is greater than the given error budget ξ, EGC picks the best node N (how to choose the best node will be introduced

[16] And we know that it many only be combined with other segments compressed by a function in LSF.

[17] $\mathscr{N}_1$ and $\mathscr{N}_2$ can be viewed as compressed segment lists.

Algorithm: Error Guarantee Computation (EGC)

Input: Query q, error budget ξ, $STree$ $\mathscr{T}_1$, $STree$ $\mathscr{T}_2$
Output: Error guarantee ε

1 $\varepsilon \leftarrow 0$;
2 Let $\mathscr{N}_1 = \{\mathscr{T}_1.root\}$ and $\mathscr{N}_2 = \{\mathscr{T}_2.root\}$ be the current accessed nodes in $\mathscr{T}_1$ and $\mathscr{T}_2$;
3 $\varepsilon \leftarrow$*ComputeErrorGuarantee*($\mathscr{N}_1, \mathscr{N}_2, q$);
4 **while** $\varepsilon > \xi$ **do**
5 $N \leftarrow$*ChooseBestNode*($\mathscr{N}_1$, $\mathscr{N}_2$, q);
6 **if** $N \in \mathscr{N}_1$ **then**
7 $\mathscr{N}_1 = \mathscr{N}_1 - \{N\} \cup \{N_l, N_r\}$;
8 **else**
9 $\mathscr{N}_2 = \mathscr{N}_2 - \{N\} \cup \{N_l, N_r\}$;
10 $\varepsilon \leftarrow$*ComputeErrorGuarantee*($\mathscr{N}_1, \mathscr{N}_2, q$);
11 Return ε;

later) to access it children nodes N_l and N_r and updates $\mathscr{N}_1$ or $\mathscr{N}_2$ (lines 5-9). Then it computes the current error guarantee based on the error measures in the current nodes (line 10). Finally, if $\varepsilon \leq \xi$, then EGC outputs ε as the final error guarantee (line 11).

Best Node Selection. The motivation of choosing one best node N is to obtain the maximal error guarantee reduction by replacing N with its children nodes N^l and N^r. Let T_N, T_{N^l} and T_{N^r} be the associated segments of N, N^l and N^r respectively. Different expressions have different guidelines. For $Sum(\boldsymbol{T_1} + \boldsymbol{T_2})$ and $Sum(\boldsymbol{T_1} - \boldsymbol{T_2})$, we choose the node maximizing $r^{\varepsilon}_{T_N} - (r^{\varepsilon}_{T_{N^l}} + r^{\varepsilon}_{T_{N^r}})$. For $Sum(\boldsymbol{T_1} \times \boldsymbol{T_2})$, we choose the node maximizing $\|\varepsilon_{T_N}\|_2 - (\|\varepsilon_{T_{N^l}}\|_2 + \|\varepsilon_{T_{N^r}}\|_2)$.

3.5.2 Incremental Error Guarantee Computation

Algorithm EGC requires to use all the currently accessed nodes to compute error guarantees at each step. Let k_1 and k_2 be the number of current accessed nodes in segment trees $\mathscr{T}_1$ and $\mathscr{T}_2$ in the current step, then the complexity of the error guarantee computation in each stage is $O(k_1 + k_2)$. It is inefficient when (i) the number of the current accesses nodes is large, and (ii) the number of executions of the error guarantee computation is large.

Table 3.10. Computation of ε_Δ

TSE	ε_Δ
$Sum(\boldsymbol{T_1} + \boldsymbol{T_2})$	$r^{\varepsilon}_{T_N} - (r^{\varepsilon}_{T_{Nl}} + r^{\varepsilon}_{Nr})$
$Sum(\boldsymbol{T_1} - \boldsymbol{T_2})$	$r^{\varepsilon}_{T_N} - (r^{\varepsilon}_{T_{Nl}} + r^{\varepsilon}_{Nr})$
$Sum(\boldsymbol{T_1} \times \boldsymbol{T_2})$	$\Big(\|\varepsilon_{T_N}\|_2(\sum_{i\in\Pi_{T_2,[a_N,b_N]}} \|\varepsilon_{T_2^i}\|_2^2)^{\frac{1}{2}}$ $-\|\varepsilon_{T_{Nl}}\|_2(\sum_{i\in\Pi_{T_2,[a_{Nl},b_{Nl}]}} \|\varepsilon_{T_2^i}\|_2^2)^{\frac{1}{2}}$ $-\|\varepsilon_{T_{Nr}}\|_2(\sum_{i\in\Pi_{T_2,[a_{Nr},b_{Nr}]}} \|\varepsilon_{T_2^i}\|_2^2)^{\frac{1}{2}}\Big)$

To improve the performance, we propose an *incremental error guarantee computation (IEGC)* optimization. The main idea of IEGC is to reuse the error guarantee returned in the previous step and the updated node N while avoiding accessing the other unchanged nodes. Applying IEGC, we are able to reduce the complexity of error guarantee computation in each step from $O(k_1 + k_2)$ to $O(1)$.

Let $\varepsilon^{(i)}$ be the error guarantee at step i, ε_Δ be the delta error guarantee computed by only using the updated node N and its children. Then the error guarantee at step $i + 1$ can be computed incrementally as $\varepsilon^{(i+1)} = \varepsilon^{(i)} - \varepsilon_\Delta$. Table 3.10 shows the computation of ε_Δ in details (assuming the chosen node N is in the STree of T_1).

3.6 Experiments

3.6.1 Environment and Setting

All experiments were conducted on a computer with a 4^{th} Intel $i7$-4770 processor (3.6 GHz), 16 GB RAM, running Ubuntu $14.04.1$. The algorithms were implemented in $C++$ and were compiled with g++ $4.8.4$.

Datasets. We evaluated all the error guarantee methods on four real-life datasets.

- Historical Forex Data (HF) [18] are tick-by-tick market data for 15 Forex (foreign exchange) data pairs, e.g., AUD/JPY (Australian Dollar vs. Japanese Yen) from May 2009 to November 2016.

[18] https://pepperstone.com/en/client-resources/historical-tick-data

Table 3.11. Data characteristics

	avg # of data points in each time series	# of time series	resolution
HF	$126,059,817$	15	millisecond
HI	$2,676,311$	14	second
HB	$1,669,835$	16	minute
HA	$1,587,258$	11	minute

Table 3.12. Number of coefficients and error measures

	# of coefficients	# of error measures
Polynomial	2	1
Gaussian	4	3

Each Forex pair is considered a time series with ~ 126 million data points (3 per second).

- Historical IoT Data (HI) were provided by Teradata and measure the internal oil pressure and the oil temperature every second from $8/19/2015$ to $11/17/2015$, as reported by seven engines in mining trucks in Chile.

- Historical Bitcoin Exchanges Data (HB)[19] contains 16 cryptocurrency exchange prices per minute from January 2012 to January 2018. Each cryptocurrency is considered as a time series.

- Historical Air Quality Data (HA)[20] present 11 different air quality measurements such as air pressure, air temperature and relative humidity from $09/10/2011$ to $09/10/2014$ in San Diego, at 1-minute resolution.

Table 3.11 summarizes the data characteristics.

The HF and HB are financial market data, which are considered hard-to-model, while HI and HA are climate data following certain patterns. For example, the temperature in afternoon

[19]https://www.kaggle.com/mczielinski/bitcoin-historical-data/data
[20]https://www.kaggle.com/ktochylin

```
SELECT TSA('(Sum((t1.timeseries-Constant(μ(t1.timeseries)))×
                 t2.timeseries-Constant(μ(t2.timeseries))))
           )/(σ(t1.timeseries)× σ(t2.timeseries))',
                 t1.timeseries, t2.timeseries)AS result
FROM HF t1, HF t2;
```

Figure 3.12. SQL query computing correlation TSA for all the time series pairs in HF.

is usually higher than that at night, etc. Not surprisingly, the HB experiments behaved very similarly to the HF experiments, while the HA experiments behaved similarly to the HI ones.

Estimation Function Families. Following the prior work lessons [43, 76], we choose the 1-degree polynomial function family ($\{ax + b|a, b \in R\}$) and the Gaussian function family ($\{a \exp\left(\frac{-(x-b)^2}{2c^2}\right) + d|a, b, c, d \in R\}$) as representatives to compress the time series. Notice that the Gaussian function family is in ANY, while the polynomial function family is in LSF (also in VS). Table 3.12 summarizes the number of coefficients and error measures stored for each segment compressed by the corresponding estimation functions.

3.6.2 Experimental Results

We evaluate the error guarantees provided by Plato on compressed segment lists and segment trees in Chapter 3.6.2.1 and Chapter 3.6.2.2 respectively.

3.6.2.1 Experimental Results on Compressed Segment Lists

We evaluate the error guarantees for TSAs over (i) aligned, fixed-length time series segmentations generated by applying the fixed-length segmentation (FL); (ii) and misaligned, variable-length time series segmentations generated by using the sliding window algorithm (SW). The details of segmentation algorithms are shown in Chapter 3.7. In order to provide a fair comparison, we fix the space cost for both cases, i.e., they have the same compression ratios.

We evaluate the correlation TSA over all the time series pairs in each dataset. The corresponding SQL queries are shown in Figure 3.12 (The SQL queries on the other three datasets are similar by changing the table HF to HI, HB and HA respectively). All the error

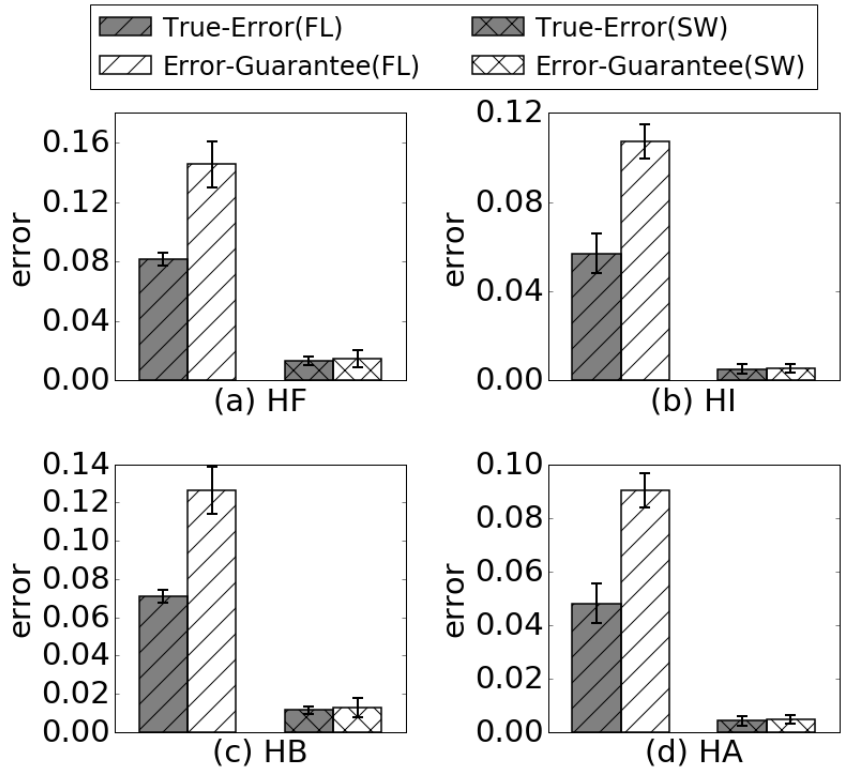


Figure 3.13. True errors and error guarantees in aligned (FL) and misaligned cases (SW). The True-Error(SW) are 0.0132 and 0.00508 in (a) and (b).

guarantees and true errors reported in the following are the average values (including the standard variances) across all correlations in a dataset.

Error Guarantees Quality. Figure 3.13 reports the absolute true errors and the error guarantees of the correlation TSAs in the aligned/fixed-length (FL) and misaligned/variable-length (SW) cases using the polynomial function family. Since the TSAs are correlations, the approximate results may range between 1 (perfect correlation) and -1 (perfect reverse correlation), with 0 meaning no correlation at all.

Under the same compression ratio [21] the variable-length error guarantees are much smaller than the fixed-length error guarantees. In Figure 3.13, the misaligned Error-Guarantee (SW) is $10\times \sim 20\times$ smaller than the aligned Error-Guarantee (FL) on the average (ranging the compression ratio from $10,000$ to 100). This is mainly because, as it has already been known,

[21] Compression ratio is the size of the original data over the size of the compressed data.

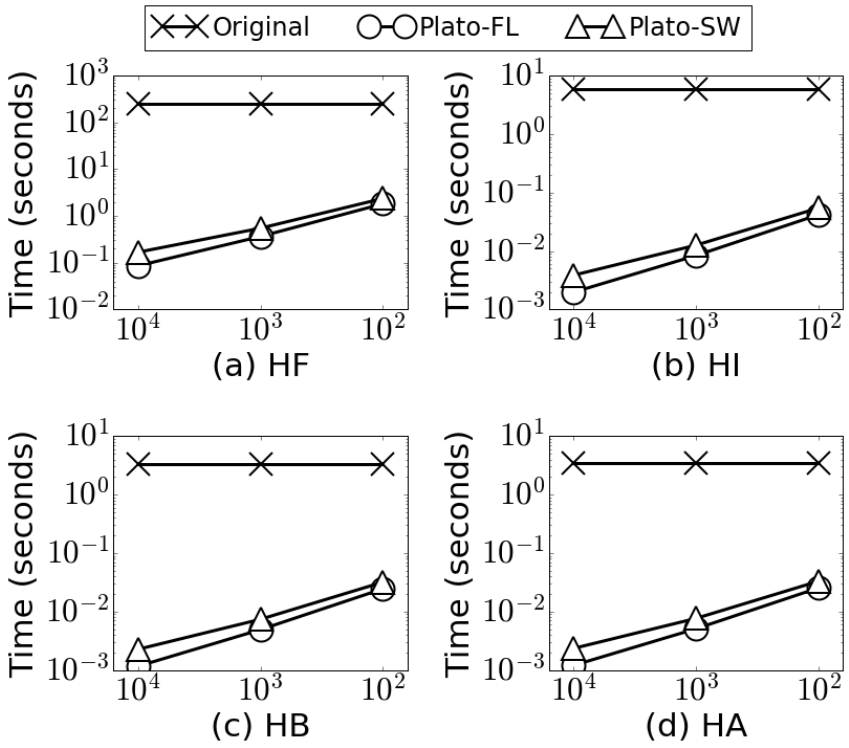


Figure 3.14. Running time of TSAs in aligned and misaligned cases.

variable-length allows for much better estimation. Indeed, notice the misaligned true errors are also much smaller than the aligned true errors. For example, In Figure 3.13, True-Error(SW) is $6\times \sim 11\times$ smaller than True-Error(FL) on the average.

Importantly, the error guarantees are close to the true errors, especially for the misaligned error guarantees, which matter most practically. In particular, Error-Guarantee(SW) is only $1.08\times \sim 1.11\times$ larger than the True-Error(SW) in HF and HI respectively (on the average). Furthermore, they are very small in absolute terms. This indicates the high quality and practicality of AI (Amplitude-independent) error guarantees.

Run Time Performance. Figure 3.14 reports the total running time of the correlation TSAs over (i) the original time series (Original), (ii) the time series segmented into a fixed length, aligned segments (Plato-FL) and (iii) time series segmented into misaligned, variable-length segments by SW (Plato-SW). The estimation function family is the polynomial family. The x-axis is the compression ratio (from 10000 to 100).

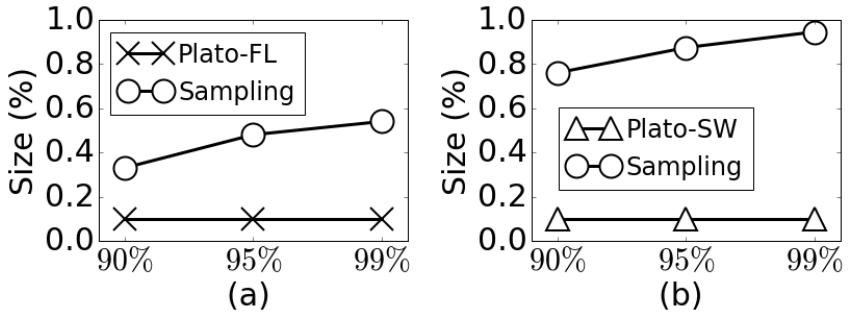


Figure 3.15. Space cost of sampling and Plato when providing the same error guarantees.

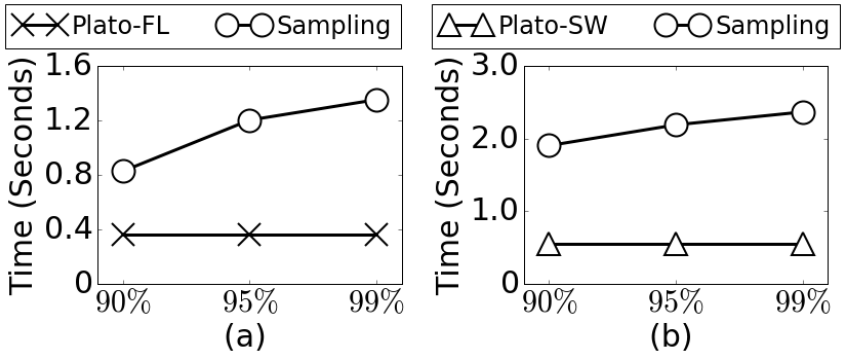


Figure 3.16. Running time of sampling and Plato when providing the same error guarantees.

Both Plato-FL and Plato-SW outperform vastly the Original in all the datasets. For example, when the compression ratio is 1000, Plato-FL and Plato-SW are about three orders of magnitude faster than Original.

Plato-SW is about $1.8\times$ slower than Plato-FL due to the intricacy of the segment combination selection algorithm. However, a mere 80% penalty is a minor price to pay for the orders-of-magnitude superior error guarantees delivered by misaligned/variable-length segmentations.

Comparison with Sampling. In this part, we compare (i) the space cost and (ii) the runtime performance of Plato with the sampling methods when providing similar error guarantees. We use a uniform random sampling scheme with a global seed in order to create a samples database. We also assume knowledge of minimums and maximums. That is, let $X_1, ..., X_n$ be

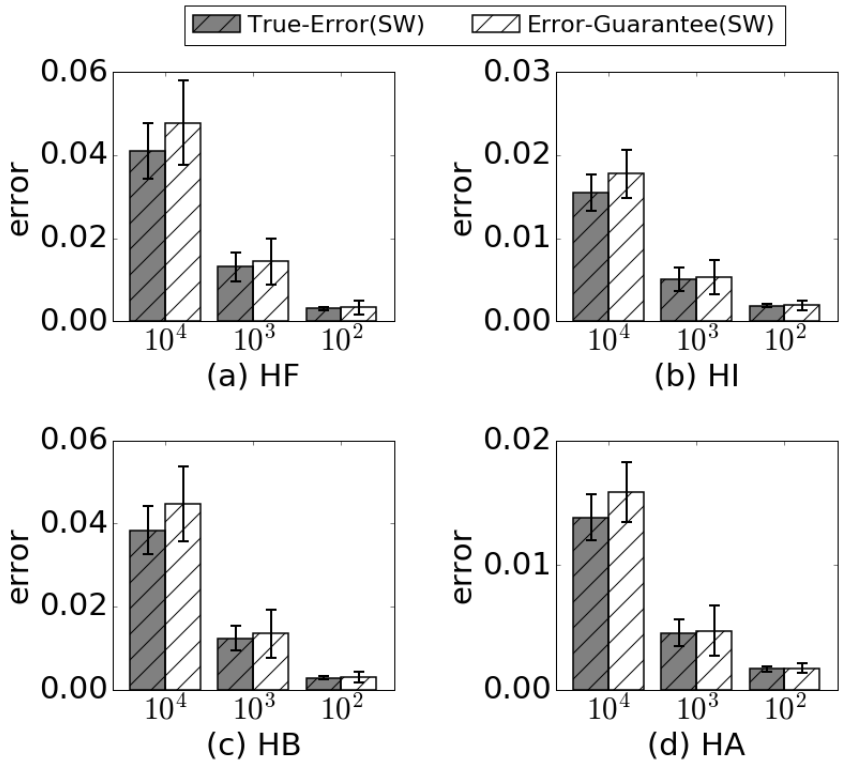


Figure 3.17. Effect of compression ratios.

the random variables such that $d_{min} \leq X_i \leq d_{max}$ for all i where $X_i = d_i^{T_1} \times d_i^{T_2}$, $d_{min} = \min\{d_i^{T_1}\} \times \min\{d_i^{T_2}\}$, and $d_{max} = \max\{d_i^{T_1}\} \times \max\{d_i^{T_2}\}$. Let $R = \sum_{i=1}^{n} X_i$ and ε be the error guarantee. Using the Chernoff bounds [35], we can obtain the minimal sample size needed in order to achieve the desired error guarantee with certain confidence.

Figure 3.15 reports the sizes (as percentage to the original data size) of sampled data points in order to provide similar error guarantees with the Plato-FL (the error guarantee of TSAs over aligned, fixed-length time series produced by FL) and Plato-SW (the error guarantee of TSAs over misaligned time series produced by SW) with 1000 compression ratio in HF respectively. Figure 3.16 shows the corresponding runtime cost. To achieve similar error guarantees, sampling needs more space and more time than Plato. We define "similar" to mean 90%, or 95% or 99% confidence - in contrast to Plato's deterministic, 100% confidence guarantees.

In the following, we study the effects of (i) compression ratios, (ii) estimation function families, (iii) orthogonal optimizations, and (iv) segment combination selection strategies.

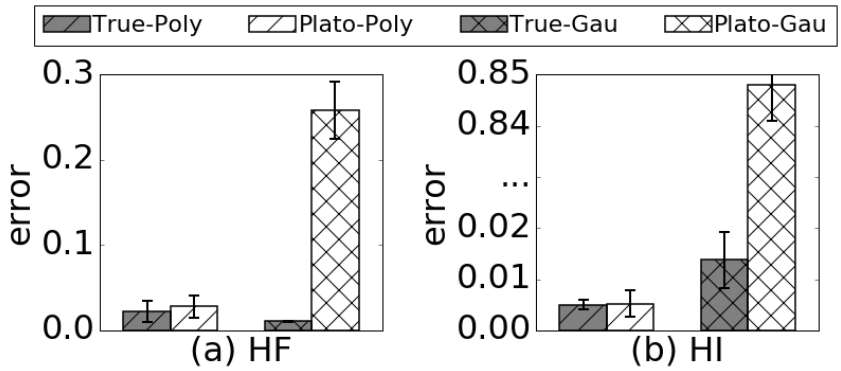


Figure 3.18. Effect of estimation function families.

Compression Ratios. In order to isolate the effect of the compression ratios, [22] we fix the estimation function family to be polynomials and fix the segment list building algorithm to be SW. In Figure 3.17, we change the compression ratios from $10,000$ to 100 by controlling the error threshold values and report the corresponding true errors (True-Error(SW)) and the error guarantees (Error-Guarantee(SW)).

Naturally, higher compression ratios lead to smaller true errors and error guarantees. For example, in Figure 3.17(a), the true error and error guarantee with 100 compression ratio are $13.32\times$ and $15.58\times$ smaller than those with $10,000$ compression ratio on the average. Importantly, the error guarantees provided by Plato are close to the true error in all the datasets and are generally small in absolute terms (with the relative exception of $10,000$ compression on HF). Again, this indicates the high quality of the error guarantees provided by Plato.

Estimation Function Families. In order to isolate the effect of the estimation function families, we fix the segment list building algorithm to be SW and fix the compression ratio to 1000. Figure 3.18 presents the true errors and the error guarantees for TSAs over time series compressed by polynomial functions (True-Error(Poly), Error-Guarantee(Poly)) and Gaussian functions (True-Error(Gau), Error-Guarantee(Gau)) respectively.

The error guarantees with estimation functions from LSF (polynomials) are significantly smaller than those with estimation functions in ANY (Gaussians). In Figure 3.18(a), Error-

[22]Compression ratio is the size of the original data over the size of the compressed data.

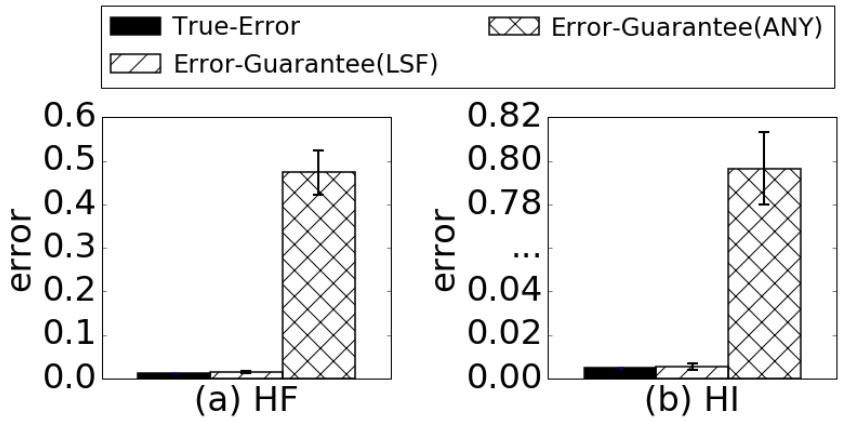


Figure 3.19. Effect of orthogonal optimization.

Guarantee(Poly) (in LSF and VS) is about $10\times$ smaller than Error-Guarantee(Gau) (in ANY) on the average and in Figure 3.18(b), Error-Guarantee(Poly) (in LSF and VS) is about $160\times$ smaller than Error-Guarantee(Gau) (in ANY) on the average. Notice that the error guarantees provided by Plato-Poly is AI, while those of Plato-Gau are not. So the results show that AI error guarantees are practical while non-AI error guarantees are not. Interestingly, True-Error(Gau) is smaller than True-Error(Poly) in the HF dataset, which indicates that Gaussian functions model HF data better than the polynomial functions - not surprising given the more random movements of financial data. The guarantees produced by the polynomials are far better thanks to AI.

Effect of Orthogonal Optimization and LSF. To measure the effect on error guarantees of the orthogonal optimization (and its extension to misaligned segmentations, enabled by LSF) we fix the estimation function family to the polynomials, which are LSF and, trivially, are also in ANY. We use both the general error guarantees of ANY (Error-Guarantee(ANY)) and the specialized error guarantees of LSF (Error-Guarantee(LSF)) for TSAs over misaligned segments compressed by polynomial functions (using variable-length segmentations with the SW algorithm). We fix the compression ratio to 1000. As shown in Figure 3.19, the error guarantee for LSF certifies that the true result is just within ± 0.0137 in HF and within ± 0.0052 in HI.

Segment Combination Selection Strategies. To isolate the quality effect of employing the optimal segment combination selection strategy (OS) we compare it with IS strategy (the

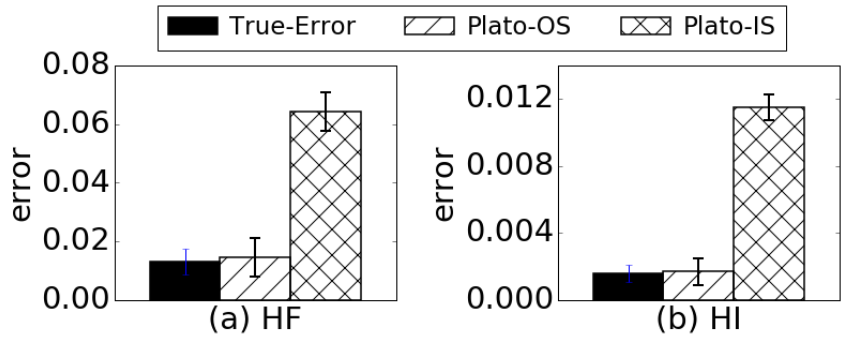


Figure 3.20. Effect of segment combination selection strategies.

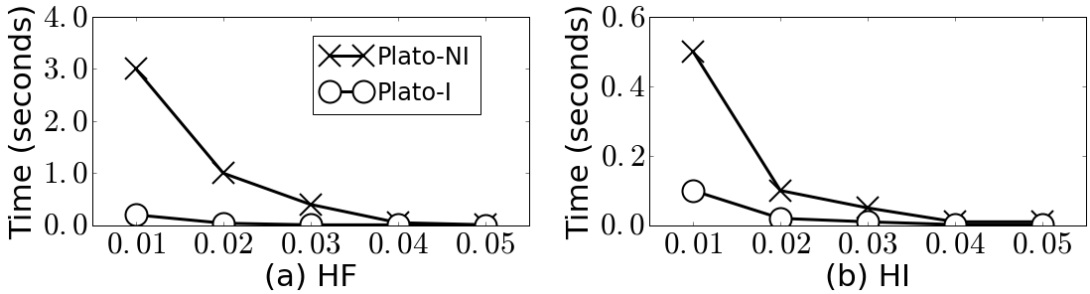


Figure 3.21. Running time with different error budgets.

straightforwad method mentioned in Chapter 3.4.3.1) on a case of variable-length compression with an LSF function family (polynomials). Figure 3.20 shows that Plato-OS is about $5\times$ smaller than Plato-MS on the average. In addition, the running time of Plato-IS and Plato-OS are close. For example, the running time of Plato-IS and Plato-OS are 0.536 and 0.548 seconds in HF respectively.

3.6.2.2 Experimental Results on Compressed Segment Tree

Run Time Performance. We evaluate the same correlation TSA (shown in Figure 3.12) on compressed segment trees with different error budgets. Plato-I (resp. Plato-NI) refers to the error guarantee computation algorithm with (resp. without) the incremental error guarantee computation optimization. According to the results in Figure 3.21, we have the following observations:

- The running time decreases as the error budget increase (for both Plato-NI and Plato-I). This

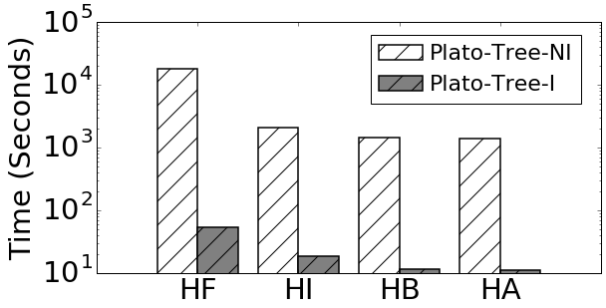


Figure 3.22. Effect of incremental segment tree building optimization.

is because Plato-NI (and Plato-I) accesses fewer nodes for queries with larger error budgets.

- Plato-I is about $5\times$ faster than Plato-NI on the average.

Segment Tree Building. We evaluate the effect of the incremental estimation function computation optimization by comparing Plato-Tree-NI with Plato-Tree-I, where Plato-Tree-NI (resp. Plato-Tree-I) refers to the segment tree building algorithm without (resp. with) incremental estimation function computation optimization. We fix the function family to be polynomials. As shown in Figure 3.22, Plato-Tree-I outperforms Plato-Tree-NI by two orders of magnitudes on the average. Notice the running time of Plato-Tree-NI on HF is more than 5 hours, we stop it after 5 hours.

3.7 Related Work

Approximate query processing (AQP) and data compression have been widely studied, whose most relevant aspects are summarized next. To the best of our knowledge, this is the first work that provides deterministic guarantees for analytics over multiple compressed time series.

AQP with Probabilistic Error Guarantees. Approximate query processing using *sampling* [15, 92, 77, 5] computes approximate answers by appropriately evaluating the queries on small samples of the data, e.g., STRAT [15], SciBORQ [92], and BlinkDB [5]. Such approaches typically leverage statistical inequalities and the central limit theorem to compute the confidence interval (or variance) of the computed approximate answer. As a result, their error guarantees are

probabilistic - as opposed to this work's deterministic (100% confidence) ones. Note however that, unlike sampling, our compression-based techniques are tuned for time series and continuous data.

AQP with Deterministic Error Guarantees. Approximately answering queries while providing deterministic error guarantees has been successfully applied in many applications [26, 34, 67, 85, 59, 84]. However, existing work in the area has focused on simple aggregation queries that involve only a single time series (or table) and aggregates such as SUM, COUNT, MIN, MAX and AVG. Our work extends the prior work, as it addresses analytics over multiple compressed time series such as correlation, cross-correlation. In addition, this work is the first one to categorize compression function families based on their suitability for error guarantees.

Data Summarizations and Compressions. Relevant work in this area has come mostly from two different communities: the database community [41, 84, 78, 95] and the signal processing community [47, 44, 43, 14, 29].

The database community has mostly focused on creating summarizations (also referred to as synopses or sketches) that can be used to answer specific queries. These include among others histograms [83, 41, 99, 89] (e.g., EquiWidth and EquiDepth histograms [83], V-Optimal histograms [41], and Hierarchical Model Fitting (HMF) histograms [99]), used among other for cardinality estimation [41] and selectivity estimation [84].

The signal processing community produced a variety of methods that can be used to compress time series data and thus are more relevant to the present work, as they provide the underlying compressions. These include among others the Piecewise Aggregate Approximation (PAA) [47], and the Piecewise Linear Representation (PLR) [43]. Plato is orthogonal to those data summarization and compression techniques.

Segmentation Algorithm. We summarize the state-of-the-art time series segmentation algorithms, which can be classified into two categories:

- Fix-length segmentation (FL), which partitions a time series based on fixed time windows.

The segments produced by the FL have equal lengths, and are utilized in our aligned-segments experiments.

- Variable-length segmentation. There are three groups of algorithms produce variable-length segmentations: the Top-down methods [61, 79], the Bottom-up approaches [48, 49] and the Sliding-window techniques [55, 45]. Among them, the Sliding-window (SW) has been proven to be more efficient than the Top-down and the Bottom-up methods [45, 46]. Thus, we choose the Sliding-window (SW) as the representative variable length segmentation algorithm in our experiments. The segments created by the SW have variable lengths [46] and are used in our misaligned-segments experiments.

STree vs. Tree-based Wavelet. The prior tree-based wavelet [88] has a similar structure compared with STree, however, they still have the following differences:

- Tree-based wavelet is a complete full binary tree, while STree may be unbalanced as it considers the smoothness of the time series by only splitting shaking segments.

- Tree-based wavelet equally partition segment, while STree chooses the best cutting position, which may not be the middle point. STree is error sensitive, while wavelet is not.

- Tree-based wavelet does not maintain error measures, while STree does, which can contribute to error guarantees in query processing.

3.8 Chapter Summary

This chapter introduces Plato, which is an approximate query processing system supporting anytime analytic queries over compressed time series with tight deterministic error guarantees. It indicates that deterministic error guarantees are feasible and practical, given the appropriate combination of error measures and estimation function family.

This chapter contains (i) material from "Plato: Approximate Queries over Compressed Time Series with Tight Deterministic Error Guarantees" by Etienne Boursier, Jaqueline J. Brito,

Chunbin Lin, and Yannis Papakonstantino, which was submitted for publication; and (ii) material from from "Supporting Anytime Queries over Compressed Time Series with Deterministic Error Guarantees" by Chunbin Lin, Joshua Lapacik, and Yannis Papakonstantino, which is in preparation for submission. The dissertation author was the primary investigator of the papers.

Chapter 4

Conclusion and Future Directions

In this dissertation, we first propose the GQ-Fast database, which is an indexed database that roughly corresponds to efficient encoding of annotated adjacency lists that combines salient features of column-based organization, indexing and compression. We used GQ-Fast to accelerate queries for two OLAP dashboards in the biomedical field. GQ-Fast outperforms PostgreSQL by $2-4$ orders of magnitude and MonetDB, Vertica and Neo4j by $1-3$ orders of magnitude when all of them are running on RAM.

Then we propose the approximate query processing system, called Plato, which supports anytime analytic queries over compressed time series. Plato allows users to input a query with TSA and an error budget, then it provides an approximate answer with a tight deterministic error guarantee, which is no greater than the given error budget. The results indicate that deterministic error guarantees are feasible and practical, given the appropriate combination of error measures and estimation function family.

Future work

There are several promising future work opportunities towards improving the applicability and the impact of the presented results.

For GQ-Fast, we will investigate how GQ-Fast can be incorporated in a general SQL processor, where GQ-Fast will execute relationship subqueries and conventional query processing

techniques will be used to combine and process the output of GQ-Fast. We will also study the pushing aggregation down optimization in order to further improve the performance.

For Plato, future work may develop such combinations for other important families also. Note that the tightness results of this paper do not preclude the future development of practical and theoretically-sound deterministic error guarantees for families are currently outside the LSF (or outside the VS in the case of aligned series). Rather, the tightness results merely state that (i) without introducing additional assumptions about a function family and/or (ii) without introducing different/additional error measures, no better error guarantees can be achieved. Researchers may come up with other interesting properties of function families outside LSF (or VS) and deliver good error guarantees, based on such properties. Finally, if some cases turn out to be unaddressable by deterministic error guarantees, another interesting area of research would be the delivery of probabilistic guarantees over compressed data. A number of applications-oriented adjustments and extensions appear possible. In such cases, the results of this work will become building blocks in more complex analysis settings. One such case is analyses that correspond to multiple queries (in contrast to this fundamentals-oriented work that considered queries that return a single number). For example, in an anomaly detection setting we would not simply be asking how much cross-correlated is the air conditioning air flow with the air temperature. We would rather ask for the time periods when the temperature decorrelated significantly from the air flow. It is easy to see how the latter can be framed as multiple queries. Of course, additional optimizations will be possible in the multiple query processing case.

Bibliography

[1] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[2] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column oriented database systems. *PVLDB*, 2(2):1664–1665, 2009.

[3] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, pages 967–980, 2008.

[4] Christopher R Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD*, 2016.

[5] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[6] Saeed Reza Aghabozorgi, Ali Seyed Shirkhorshidi, and Ying Wah Teh. Time-series clustering - A decade review. *Inf. Syst.*, 53:16–38, 2015.

[7] Yanif Ahmad and Christoph Koch. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*, 2(2):1566–1569, 2009.

[8] Gennady Antoshenkov. Byte-aligned bitmap compression. In *DCC*, page 476, 1995.

[9] Åke Björck. Solving linear least squares problems by gram-schmidt orthogonalization. *BIT Numerical Mathematics*, 7(1):1–21, 1967.

[10] Peter Bloomfield. *Fourier analysis of time series: an introduction*. John Wiley & Sons, 2004.

[11] Subhrajyoti Bordoloi and Bichitra Kalita. Designing graph database models from existing relational databases. *IJCA*, 74(1), 2013.

[12] Jaqueline Brito, Korhan Demirkaya, Boursier Etienne, Yannis Katsis, Chunbin Lin, and Yannis Papakonstantinou. Efficient approximate query answering over sensor data with deterministic error guarantees. *CoRR*, abs/1707.01414, 2017.

[13] John Catozzi and Sorana Rabinovici. Operating system extensions for the teradata parallel VLDB. In *VLDB*, pages 679–682, 2001.

[14] Kin-pong Chan and Ada Wai-Chee Fu. Efficient time series matching by wavelets. In *ICDE*, pages 126–133, 1999.

[15] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. Optimized stratified sampling for approximate query processing. *TODS*, 32(2):9, 2007.

[16] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[17] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. Approximate query processing: no silver bullet. In *Sigmod*, pages 511–519. ACM, 2017.

[18] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and S Yu Philip. Graph OLAP: a multi-dimensional framework for graph data analysis. *KAIS*, 21(1):41–63, 2009.

[19] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and Philip S Yu. Graph OLAP: Towards online analytical processing on graphs. In *ICDM*, pages 103–112, 2008.

[20] Lei Chen and Raymond T. Ng. On the marriage of lp-norms and edit distance. In *VLDB*, pages 792–803, 2004.

[21] Peter Chen. Entity-relationship modeling: historical events, future trends, and lessons learned. In *Software pioneers*, pages 296–310. Springer, 2002.

[22] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. *ACM SIGMOD Record*, 30(2):271–282, 2001.

[23] Ward Cheney and David Kincaid. Linear algebra: Theory and applications. *The Australian Mathematical Society*, 110, 2009.

[24] ByoungSeon Choi. *ARMA model identification*. Springer Science & Business Media, 2012.

[25] Kuo-Liang Chung and Jung-Gen Wu. Level-compressed huffman decoding. *TCOM*, 47(10):1455–1457, 1999.

[26] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Effective computation of biased quantiles over data streams. In *ICDE*, pages 20–31, 2005.

[27] DGT Denison, BK Mallick, and AFM Smith. Automatic bayesian curve fitting. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 60(2):333–350, 1998.

[28] Andrew Eisenberg and Jim Melton. Sql/xml is making good progress. *ACM Sigmod Record*, 31(2):101–108, 2002.

[29] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, 1994.

[30] Edward J Franczek, John Thomas Bretscher, and Raymond Walden Bennett III. Computer virus screening methods and systems, November 16 1999. US Patent 5,987,610.

[31] Alex Galakatos, Andrew Crotty, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. Revisiting reuse for approximate query processing. *PVLDB*, 10(10):1142–1153, 2017.

[32] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[33] Goetz Graefe. Query evaluation techniques for large databases. *CSUR*, 25(2):73–169, 1993.

[34] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD*, pages 58–66, 2001.

[35] Torben Hagerup and Christine Rüb. A guided tour of chernoff bounds. *Information processing letters*, 33(6):305–308, 1990.

[36] Paul Richard Halmos. *Finite-dimensional vector spaces*. Springer Science & Business Media, 2012.

[37] Walter Hoffmann. Iterative algorithms for gram-schmidt orthogonalization. *Computing*, 41(4):335–348, 1989.

[38] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: a DSL for easy and efficient graph analysis. In *ISCA*, volume 40, pages 349–362. ACM, 2012.

[39] Vagelis Hristidis, Yannis Papakonstantinou, and Andrey Balmin. Keyword proximity search on XML graphs. In *ICDE*, pages 367–378. IEEE, 2003.

[40] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *DEBU*, 35(1):40–45, 2012.

[41] Yannis E. Ioannidis and Viswanath Poosala. Balancing histogram optimality and practicality for query result size estimation. In *SIGMOD*, pages 233–244, 1995.

[42] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. Trends in big data analytics. *Journal of Parallel and Distributed Computing*, 74(7):2561–2573, 2014.

[43] Eamonn Keogh. Fast similarity search in the presence of longitudinal scaling in time series databases. In *ICTAI*, pages 578–584, 1997.

[44] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. *SIGMOD Record*, 30(2):151–162, 2001.

[45] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. An online algorithm for segmenting time series. In *ICDM*, pages 289–296, 2001.

[46] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. Segmenting time series: A survey and novel approach. In *Data mining in time series databases*, pages 1–21. World Scientific, 2004.

[47] Eamonn J. Keogh, Kaushik Chakrabarti, Michael J. Pazzani, and Sharad Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *KAIS*, 3(3):263–286, 2001.

[48] Eamonn J. Keogh and Michael J. Pazzani. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In *KDD*, pages 239–243, 1998.

[49] Eamonn J Keogh and Michael J Pazzani. Relevance feedback retrieval of time series data. In *SIGIR*, pages 183–190, 1999.

[50] Halil Kilicoglu, Marcelo Fiszman, Alejandro Rodriguez, Dongwook Shin, A Ripple, and Thomas C Rindflesch. Semantic MEDLINE: a web application for managing the results of PubMed searches. In *SMBM*, volume 2008, pages 69–76, 2008.

[51] Halil Kilicoglu, Dongwook Shin, Marcelo Fiszman, Graciela Rosemblat, and Thomas C Rindflesch. SemMedDB: a PubMed-scale repository of biomedical semantic predications. *Bioinformatics*, 28(23):3158–3160, 2012.

[52] Kyoung-jae Kim. Financial time series forecasting using support vector machines. *Neurocomputing*, 55(1-2):307–319, 2003.

[53] James Knight. Method and system for remote network security management, April 28 2004. US Patent App. 10/834,443.

[54] Donald E Knuth. Dynamic huffman coding. *Journal of algorithms*, 6(2):163–180, 1985.

[55] Antti Koski, Martti Juhola, and Merik Meriste. Syntactic recognition of ecg signals by attributed finite automata. *Pattern Recognition*, 28(12):1927–1940, 1995.

[56] Geza Kovács, Shay Zucker, and Tsevi Mazeh. A box-fitting algorithm in the search for periodic transits. *Astronomy & Astrophysics*, 391(1):369–377, 2002.

[57] Arun Kumar, Robert McCann, Jeffrey F. Naughton, and Jignesh M. Patel. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Record*, 44(4):17–22, 2015.

[58] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The Vertica analytic database: C-store 7 years later. *VLDB*, 5(12):1790–1801, 2012.

[59] Iosif Lazaridis and Sharad Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD*, pages 401–412, 2001.

[60] Iosif Lazaridis and Sharad Mehrotra. Capturing sensor-generated time series with quality guarantees. In *ICDE*, pages 429–440, 2003.

[61] Chung-Sheng Li, Philip S. Yu, and Vittorio Castelli. MALM: A framework for mining sequence database at multiple abstraction levels. In *CIKM*, pages 267–272, 1998.

[62] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. Hippogriffdb: Balancing I/O and GPU bandwidth in big data analytics. *PVLDB*, 9(14):1647–1658, 2016.

[63] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. Fast in-memory SQL analytics on typed graphs. *PVLDB*, 10(3):265–276, 2016.

[64] Chunbin Lin, Jianguo Wang, and Yannis Papakonstantinou. GQFast: Fast graph exploration with context-aware autocompletion. In *ICDE*, pages 1389–1390, 2017.

[65] Hongbin Ma, Bin Shao, Yanghua Xiao, Liang Jeff Chen, and Haixun Wang. G-SQL: Fast query processing via graph exploration. *PVLDB*, 9(12), 2016.

[66] Stefan Manegold, Martin L Kersten, and Peter Boncz. Database architecture evolution: mammals flourished long before dinosaurs became extinct. *PVLDB*, 2(2):1648–1653, 2009.

[67] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *SIGMOD*, pages 426–435, 1998.

[68] Jonathan Mei and José M. F. Moura. Signal processing on graphs: Causal modeling of unstructured data. *IEEE Trans. Signal Processing*, 65(8):2077–2092, 2017.

[69] Michael D. Morse and Jignesh M. Patel. An efficient and accurate method for evaluating time series similarity. In *SIGMOD*, pages 569–580, 2007.

[70] Brandon Myers, Jeremy Hyrkas, Daniel Halperin, and Bill Howe. Compiled plans for in-memory path-counting queries. In *IMDM@VLDB*, pages 28–43. 2015.

[71] Gonzalo Navarro and Nieves Brisaboa. New bounds on D-ary optimal codes. *Information Processing Letters*, 96(5):178–184, 2005.

[72] Edward Nelson. Probability theory and euclidean field theory. In *Constructive quantum field theory*, pages 94–124. Springer, 1973.

[73] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[74] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471. ACM, 2013.

[75] Sriram Padmanabhan, Timothy Malkemus, Anant Jhingran, and Ramesh Agarwal. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, pages 567–574, 2001.

[76] Zhuokun Pan, Yueming Hu, and Bin Cao. Construction of smooth daily remote sensing time series data: a higher spatiotemporal resolution perspective. *Open Geospatial Data, Software and Standards*, 2(1):25, 2017.

[77] Niketan Pansare, Vinayak R. Borkar, Chris Jermaine, and Tyson Condie. Online aggregation for large mapreduce jobs. *PVLDB*, 4(11):1135–1145, 2011.

[78] Odysseas Papapetrou, Minos N. Garofalakis, and Antonios Deligiannakis. Sketch-based querying of distributed sliding-window data streams. *PVLDB*, 5(10):992–1003, 2012.

[79] Sanghyun Park, Dongwon Lee, and Wesley W Chu. Fast retrieval of similar subsequences in long sequence databases. In *KDEX*, pages 60–67, 1999.

[80] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178. ACM, 2009.

[81] Donald B Percival and Andrew T Walden. *Wavelet methods for time series analysis*, volume 4. Cambridge university press, 2006.

[82] John S Philo. An improved function for fitting sedimentation velocity data for low-molecular-weight solutes. *Biophysical Journal*, 72(1):435–444, 1997.

[83] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In *SIGMOD*, pages 256–276, 1984.

[84] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD*, pages 294–305, 1996.

[85] Navneet Potti and Jignesh M. Patel. DAQ: A new paradigm for approximate query processing. *PVLDB*, 8(9):898–909, 2015.

[86] K Priyanka and Nagarathna Kulennavar. A survey on big data analytics in health care. *International Journal of Computer Science and Information Technologies*, 5(4):5865–5868, 2014.

[87] Wullianallur Raghupathi and Viju Raghupathi. Big data analytics in healthcare: promise and potential. *Health information science and systems*, 2(1):3, 2014.

[88] Idan Ram, Michael Elad, and Israel Cohen. Generalized tree-based wavelet transform. *IEEE Transactions on Signal Processing*, 59(9):4199–4209, 2011.

[89] Frederick Reiss, Minos N. Garofalakis, and Joseph M. Hellerstein. Compact histograms for hierarchical identifiers. In *VLDB*, pages 870–881, 2006.

[90] Maayan Roth, Assaf Ben-David, David Deutscher, Guy Flysher, Ilan Horn, Ari Leichtberg, Naty Leiser, Yossi Matias, and Ron Merom. Suggesting friends using the implicit social graph. In *ACM SIGKDD*, pages 233–242, 2010.

[91] Mark A Roth and Scott J Van Horn. Database compression. *ACM Sigmod Record*, 22(3):31–39, 1993.

[92] Lefteris Sidirourgos, Martin L. Kersten, and Peter A. Boncz. Sciborq: Scientific data management with bounds on runtime and quality. In *CIDR*, pages 296–301, 2011.

[93] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-june Paul Hsu, and Kuansan Wang. An overview of Microsoft Academic Service (MAS) and applications. In *WWW*, pages 243–246. ACM, 2015.

[94] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[95] Daniel Ting. Towards optimal cardinality estimation of unions and intersections with sketches. In *SIGKDD*, pages 1195–1204, 2016.

[96] Mikio Tobita. Combined logarithmic and exponential function model for fitting post-seismic gnss time series after 2011 tohoku-oki earthquake. *Earth, Planets and Space*, 68(1):41, 2016.

[97] Panos Vassiliadis and Timos Sellis. A survey of logical models for OLAP databases. *ACM Sigmod Record*, 28(4):64–69, 1999.

[98] Michail Vlachos, Dimitrios Gunopulos, and George Kollios. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.

[99] Hai Wang and Kenneth C. Sevcik. Histograms based on the minimum description length principle. *VLDB J.*, 17(3):419–442, 2008.

[100] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. An experimental study of bitmap compression vs. inverted list compression. In *SIGMOD*, pages 993–1008, 2017.

[101] Dewi W Wardani and Josef Kiing. Semantic mapping relational to graph model. In *IC3INA*, pages 160–165. IEEE, 2014.

[102] WJ Wiscombe and JW Evans. Exponential-sum fitting of radiative transmission functions. *Journal of Computational Physics*, 24(4):416–444, 1977.

[103] Fan Xia, Ye Li, Chengcheng Yu, Haixin Ma, and Weining Qian. Bsma: A benchmark for analytical queries over social media data. *VLDB*, 7(13):1573–1576, 2014.

[104] Zhuoming Xu, Shichao Zhang, and Yisheng Dong. Mapping between relational database schema and OWL ontology for deep annotation. In *WI*, pages 548–552. IEEE, 2006.

[105] Shufeng Zhou. Exposing relational database as RDF. In *IIS*, volume 2, pages 237–240. IEEE, 2010.

[106] Marcin Zukowski, Peter A Boncz, Niels Nes, and Sándor Héman. MonetDB/X100 – a DBMS in the CPU cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.