DISTRIBUTED REAL-TIME ANALYSIS OF HIGH-VELOCITY MANY-DIMENSIONAL DATA SETS

by

David E. Robillard

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

at

Carleton University
Ottawa, Ontario
February 2018

© Copyright by David E. Robillard, 2018
Abstract

Fast aggregation of data with many hierarchical dimensions is a key component of many data analysis applications. Recent years have seen increased interest in the ability to do so in real-time, allowing live analysis of rapidly changing data sets. At the same time, data sets of interest have increased in size well beyond the capabilities of a single machine to process efficiently. With the widespread availability of virtualized cloud computing environments, organizations increasingly demand systems that can be deployed on commodity platforms.

Though each of these requirements in isolation can be addressed with well-known solutions, their combination introduces several system design challenges. As a result, users are forced to sacrifice certain functionality, or use several systems to handle different uses of the same data set.

This dissertation proposes a scalable distributed system, VelocityOLAP (VOLAP), which meets all of these requirements at once. Specifically, the system can rapidly ingest data while answering queries which aggregate large fractions of the database. Users are not required to “slice” along particular dimensions or generate separate indices in order to query the database. Data is distributed across many machines, and the system can grow or shrink dynamically to accommodate changing load. The distribution of data is dynamically adjusted to maintain performance and provide fault-tolerance without interrupting real-time processing. The system can provide strong guarantees that query results are fresh, even as new items are inserted at a high rate.

These results are achieved by building the system around a spatial index data structure which is specially designed to store hierarchical multi-dimensional data with rapid ingestion. Despite the lack of any user tuning for specific query workloads, extensive benchmarks on diverse data sets confirm that VOLAP scales well and can maintain high performance for a wide range of queries on a large and rapidly growing database.
Acknowledgements

A PhD is a long and difficult, if rewarding, undertaking. Though I could never acknowledge everyone who has helped me in this process within the confines of this page, I would like to thank those who have directly contributed to this work here.

First and foremost, I must thank my advisor Frank Dehne. Without his guidance and keen eye for fruitful areas of research, this project would not have been possible.

The work in this dissertation is the result of collaboration with several others. In particular, Andrew Rau-Chaplin and Neil Burke made significant contribution to the development and refinement of the ideas here. Portions of the implementation are derived from earlier work by John Caskey, Quan Kong, Hamidreza Zaboli, and Xiaoyun Zhou.

I would also like to thank the IBM Center for Advanced Studies Canada and the Natural Sciences and Engineering Research Council of Canada for providing funding that contributed to this work.

Finally, I would like to thank my family and friends for their support, without which none of this would have been possible.
# Table of Contents

Abstract ................................................. ii

Acknowledgements .......................................... iii

List of Figures ............................................. x

List of Algorithms .......................................... xi

List of Theorems ............................................ xii

List of Definitions .......................................... xiii

1 Introduction ............................................. 1

1.1 Motivation ............................................. 1

1.2 Limitations of Previous Solutions ......................... 2

1.3 Summary of Contributions ................................ 3

1.3.1 Publications ......................................... 4

1.4 Organization of Dissertation .................... 5

2 Background .................................................. 6

2.1 On-Line Analytical Processing (OLAP) ................... 6

2.1.1 Dimension Hierarchies ................................. 7

2.1.2 OLAP Operations ................................... 7

2.2 Spatial Index Trees ................................... 10

2.3 The Hilbert Curve .................................... 12

3 Model and Representation .................................. 14

3.1 Application Model .................................... 14

3.1.1 Operations ......................................... 15

3.1.2 Consistency ......................................... 17

3.2 Hierarchical Identifiers (IDs) ......................... 19

3.2.1 Representation .................................... 20

3.2.2 Operations ......................................... 22

3.3 Minimum Describing Sets (MDSs) ..................... 23

3.3.1 Representation .................................... 25

3.3.2 Operations ......................................... 26

3.4 Summary of Common Notation .................... 30

4 The Hilbert PDC-tree .................................... 31

4.1 Introduction ............................................. 31

4.2 Related Work ........................................... 32
7.1.6 Practical Application ........................................ 137

A MDS Representation ............................................. 138

B Synthetic Data Generation ................................. 141

  B.1 Generating Skewed Hierarchical Identifiers ................. 142
  B.2 Generating Queries ......................................... 144

Bibliography .................................................. 147
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>An OLAP data cube and derived views</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Hierarchical dimensions for sales from the TPC-DS data set</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>OLAP slice operation</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>OLAP dice operation</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>OLAP roll-up operation on a single dimension</td>
<td>8</td>
</tr>
<tr>
<td>2.6</td>
<td>OLAP roll-up operation which produces a single value</td>
<td>9</td>
</tr>
<tr>
<td>2.7</td>
<td>An R-tree for a set of 8 points</td>
<td>11</td>
</tr>
<tr>
<td>2.8</td>
<td>First three iterations of a 2D Hilbert curve construction</td>
<td>12</td>
</tr>
<tr>
<td>2.9</td>
<td>Third approximation of a 2D Hilbert curve with discrete coordinates</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Real-time OLAP application model</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Hierarchical ID structure</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>An MBR and MDS for a set of points</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>Flat MDS representation</td>
<td>25</td>
</tr>
<tr>
<td>3.5</td>
<td>Finding the coverage of two MDSs</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>IDs in the DC-tree</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>A DC-tree for a set of 8 points</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>A Hilbert PDC-tree for a set of 8 points</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Number of bits used for various Hilbert mappings</td>
<td>39</td>
</tr>
<tr>
<td>4.5</td>
<td>Overlap at each split point in observed directory nodes</td>
<td>42</td>
</tr>
<tr>
<td>4.6</td>
<td>Split index frequency and overlap with fixed fanout</td>
<td>43</td>
</tr>
<tr>
<td>4.7</td>
<td>Split index frequency and overlap with supernodes</td>
<td>43</td>
</tr>
<tr>
<td>4.8</td>
<td>A node with no local minimum split</td>
<td>44</td>
</tr>
<tr>
<td>4.9</td>
<td>Tree insertion step 1: descend and update</td>
<td>47</td>
</tr>
<tr>
<td>4.10</td>
<td>Tree insertion step 2: split parent</td>
<td>48</td>
</tr>
<tr>
<td>4.11</td>
<td>Tree insertion step 3: split grandparent</td>
<td>48</td>
</tr>
<tr>
<td>4.12</td>
<td>Tree insertion step 4: split root</td>
<td>48</td>
</tr>
<tr>
<td>4.13</td>
<td>Performance for a stream of inserts</td>
<td>55</td>
</tr>
<tr>
<td>4.14</td>
<td>Performance for a stream of queries</td>
<td>56</td>
</tr>
<tr>
<td>4.15</td>
<td>Performance for a mixed stream of inserts and queries</td>
<td>57</td>
</tr>
<tr>
<td>4.16</td>
<td>Aggregate query latency for various coverages</td>
<td>57</td>
</tr>
<tr>
<td>4.17</td>
<td>Speedup for a mixed stream of inserts and queries</td>
<td>58</td>
</tr>
<tr>
<td>4.18</td>
<td>Latency as number of dimensions is increased</td>
<td>60</td>
</tr>
<tr>
<td>5.1</td>
<td>System overview</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>Server index data structure</td>
<td>67</td>
</tr>
<tr>
<td>5.3</td>
<td>An insertion with $N_r = 3, W = 2$</td>
<td>69</td>
</tr>
<tr>
<td>5.4</td>
<td>Transitions within an operation stream</td>
<td>71</td>
</tr>
<tr>
<td>5.5</td>
<td>System overview with send thread for serializable execution</td>
<td>71</td>
</tr>
<tr>
<td>5.6</td>
<td>Path of a request through the system with serialization points</td>
<td>72</td>
</tr>
</tbody>
</table>
B.2 Zipf-distributed hierarchical IDs with smooth skew . . . . . . . . . . . 144
List of Algorithms

<table>
<thead>
<tr>
<th>Section</th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>mds_contains</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>mds_coverage</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>select_parent_node</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>tree_insert</td>
<td>41</td>
</tr>
<tr>
<td>4.3</td>
<td>split_node</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>split_up</td>
<td>46</td>
</tr>
<tr>
<td>4.5</td>
<td>split_root</td>
<td>47</td>
</tr>
<tr>
<td>4.6</td>
<td>tree_query</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>iq_barrier_start_read</td>
<td>73</td>
</tr>
<tr>
<td>5.2</td>
<td>iq_barrier_finish_read</td>
<td>73</td>
</tr>
<tr>
<td>5.3</td>
<td>serialize_read</td>
<td>74</td>
</tr>
<tr>
<td>5.4</td>
<td>server_insert</td>
<td>75</td>
</tr>
<tr>
<td>5.5</td>
<td>worker_insert</td>
<td>76</td>
</tr>
<tr>
<td>5.6</td>
<td>server_range_query</td>
<td>78</td>
</tr>
<tr>
<td>5.7</td>
<td>worker_range_query</td>
<td>78</td>
</tr>
<tr>
<td>B.1</td>
<td>gen_query</td>
<td>145</td>
</tr>
</tbody>
</table>
List of Theorems

<table>
<thead>
<tr>
<th>Section</th>
<th>Theorem/Lemma Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Theorem (ID subsumption ordering)</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Lemma (MDS contains time complexity)</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>Lemma (MDS coverage time complexity)</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Lemma (Node selection lock order)</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>Lemma (Split node lock order)</td>
<td>51</td>
</tr>
<tr>
<td>4.3</td>
<td>Lemma (Split up lock order)</td>
<td>51</td>
</tr>
<tr>
<td>4.4</td>
<td>Lemma (Tree insert lock order)</td>
<td>52</td>
</tr>
<tr>
<td>4.5</td>
<td>Lemma (Tree query lock order)</td>
<td>52</td>
</tr>
<tr>
<td>4.6</td>
<td>Theorem (Tree deadlock freedom)</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>Lemma (Tree insert consistency)</td>
<td>53</td>
</tr>
<tr>
<td>4.8</td>
<td>Theorem (Tree consistency)</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>Theorem (VOLAP serializable execution)</td>
<td>87</td>
</tr>
</tbody>
</table>
# List of Definitions

<table>
<thead>
<tr>
<th>Section</th>
<th>Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Minimum Bounding Rectangle (MBR)</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Hilbert index</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Point</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Measure</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Data item</td>
<td>15</td>
</tr>
<tr>
<td>3.4</td>
<td>Insert</td>
<td>16</td>
</tr>
<tr>
<td>3.5</td>
<td>Box</td>
<td>16</td>
</tr>
<tr>
<td>3.6</td>
<td>Aggregate query</td>
<td>16</td>
</tr>
<tr>
<td>3.7</td>
<td>Coverage</td>
<td>16</td>
</tr>
<tr>
<td>3.8</td>
<td>Serializable</td>
<td>17</td>
</tr>
<tr>
<td>3.9</td>
<td>Eventual consistency</td>
<td>18</td>
</tr>
<tr>
<td>3.10</td>
<td>Staleness</td>
<td>18</td>
</tr>
<tr>
<td>3.11</td>
<td>Level</td>
<td>19</td>
</tr>
<tr>
<td>3.12</td>
<td>ID</td>
<td>19</td>
</tr>
<tr>
<td>3.13</td>
<td>Ancestor</td>
<td>19</td>
</tr>
<tr>
<td>3.14</td>
<td>Subsumes</td>
<td>20</td>
</tr>
<tr>
<td>3.15</td>
<td>Minimum Describing Set (MDS)</td>
<td>24</td>
</tr>
<tr>
<td>3.16</td>
<td>Contains</td>
<td>24</td>
</tr>
<tr>
<td>3.17</td>
<td>Coverage</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>Split descendant, split ancestor</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Node version</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>Node lock ordering</td>
<td>50</td>
</tr>
<tr>
<td>5.1</td>
<td>Workload Mix</td>
<td>90</td>
</tr>
<tr>
<td>6.1</td>
<td>Data stream DATA(n, f, D)</td>
<td>104</td>
</tr>
<tr>
<td>6.2</td>
<td>$(t, k)$-staleness</td>
<td>104</td>
</tr>
<tr>
<td>6.3</td>
<td>True aggregate value</td>
<td>107</td>
</tr>
<tr>
<td>6.4</td>
<td>Observed aggregate value</td>
<td>108</td>
</tr>
<tr>
<td>6.5</td>
<td>Aggregate relative error</td>
<td>108</td>
</tr>
<tr>
<td>6.6</td>
<td>$(t, \varepsilon)$-staleness</td>
<td>109</td>
</tr>
<tr>
<td>6.7</td>
<td>Bounded $(t, k)$-staleness</td>
<td>109</td>
</tr>
<tr>
<td>6.8</td>
<td>Bounded $(t, \varepsilon)$-staleness</td>
<td>110</td>
</tr>
<tr>
<td>6.9</td>
<td>Monotonically increasing aggregation function</td>
<td>115</td>
</tr>
<tr>
<td>6.10</td>
<td>Monotonically decreasing aggregation function</td>
<td>115</td>
</tr>
<tr>
<td>B.1</td>
<td>Data Skew</td>
<td>141</td>
</tr>
<tr>
<td>B.2</td>
<td>Zipf Distribution</td>
<td>141</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

1.1 Motivation

Many applications rely on the ability to quickly aggregate “slices” of multi-dimensional data sets with dimension hierarchies. This is particularly true in the field of data analytics, where there has been increasing interest in the ability to do so in real-time. The motivating use case is the ability to perform live analysis of high-velocity data streams along with their history, so that query results may include many gigabytes of historical data as well as items that were inserted only milliseconds prior to the query.

Unlike streaming applications which dispose of input once it is processed, such applications inherently require a large amount of storage if the rate of ingestion is high. These storage and processing requirements can easily surpass the capabilities of a single machine. Consequently, these applications require a distributed system which can spread the load among as many machines as necessary.

A system that is able to dynamically scale out to more machines on an as-needed basis could address these needs in a flexible and cost-effective way, since only the computational resources necessary at a given time would need to be allocated. If such adjustments could be made without interrupting processing, users could rely on real-time analysis being available regardless of changing system load, network architecture, or data size. Similarly, a fully decentralized architecture would allow the system to remain available in the case of hardware or network failures. These concerns are increasingly important across fields as diverse as business management [55] and health care [64].

In addition to improving existing applications that perform offline or delayed analysis, such a system would enable new applications that use up-to-date historical analysis in domains where this was previously infeasible, such as large physical sensor networks and Internet-scale system and traffic monitoring.

1. In this context, “real time” refers to interactive use, where an analyst can interactively explore the data without waiting an unreasonable amount of time for results.
1.2 Limitations of Previous Solutions

Traditional OLAP systems often take the static data cube approach [35] and materialize some or all of the multi-dimensional views to ensure high query performance. However, such systems can only be updated periodically in batches, e.g. once every week, which prevents queries from including the most recent data. This reflects the traditional distinction between OLTP and OLAP systems: transactions are processed by one system, and analysis is performed on another, typically called the data warehouse. Since the data warehouse can only be updated periodically, this separation is limiting for analysts who wish to work with the most recent data available [62].

Some more recent systems take a more dynamic approach to avoid fully materializing the necessary views, essentially allowing a lightweight data warehouse to be created on demand. This can significantly reduce the age of the most recent data available for analysis, but this model is not ideal for real-time applications where many insertions and queries are performed in tandem.

The well-known “5 V’s of Big Data” (volume, velocity, variety, veracity, value) highlight the importance of processing large data sets that have a high rate of change, or velocity. As data velocity increases, out of date results become increasingly problematic, and the approach of maintaining a separate data warehouse becomes impractical. Applications that continuously monitor new events in high-velocity data streams require the ability to analyze new data as it arrives, in real-time.

Many systems have addressed performing analysis on very large data sets with a slow rate of change. Others have addressed processing high-velocity ephemeral streams where only very recent items are relevant. However, few have addressed the union: performing up-to-date analysis on very large data sets which are constantly updated by a high-velocity stream of changes. In particular, to the author’s knowledge, no existing system can address both of these concerns on a fully decentralized and elastic architecture, while providing the strong execution order guarantees expected of transactional systems, and the ability to perform very large aggregations with suitable performance for real-time use.

2. On-Line Transaction Processing and On-Line Analytical Processing, respectively.
1.3 Summary of Contributions

This work aims to show that it is possible for a single system to:

- Ingest new data items at a high velocity.
- Aggregate large fractions of the data quickly enough for real-time use.
- Support many hierarchical dimensions, and queries specified at any granularity.
- Perform well for diverse data sets and queries, without “special” dimensions.
- Use a fully decentralized elastic architecture capable of adapting to changing load and hardware failure without interrupting processing.
- Provide strong serialization which guarantees that all inserts previously issued by a user are included in query results for that user.
- Provide query freshness between users that is suitable for real-time use.

The system described here, VOLAP, meets all of these requirements, and has been experimentally shown to perform and scale well. For example, using 20 Amazon EC2 c3.4xlarge with 1 billion items from the TPC-DS test suite, VOLAP is able to ingest over 600 thousand items per second, and process interspersed insertions and aggregate queries at approximately 50 thousand inserts and 20 thousand queries per second. With another data set, VOLAP has been shown to be capable of ingesting well over 1 million items per second.

These results are achieved by building the system around a novel data structure, the Hilbert PDC-tree, which is designed to rapidly ingest and query multi-dimensional data. Experiments show that the Hilbert PDC-tree can ingest data over 17 times faster than its predecessor, the PDC-tree, while providing comparable, and often significantly better, query performance. Experiments also show that the Hilbert PDC-tree is better able to scale to a high number of dimensions.

Continuous availability is achieved by carefully designing all operations to correctly handle a continuous incoming stream of requests in parallel without interruption, including during data migration.

VOLAP can guarantee strong serialization, though this strict guarantee is only provided within a single client’s operation stream. However, experiments show that, between users, VOLAP provides suitably fresh query results for real-time use. For example, a typical
configuration shows nearly 100% probability of perfectly fresh queries after only 15 ms. The impact of system configuration on query staleness has been evaluated by extending previous models to be applicable to aggregate queries. Experiments show that this approach models real system behaviour with high accuracy, which allows the trade-off between staleness and performance to be inexpensively explored in order to determine an optimal system configuration for a given set of requirements. This novel model is sufficiently general to be capable of analyzing and predicting the behaviour of systems other than VOLAP.

In summary, this work demonstrates that a real-time OLAP system can meet all of the above requirements, and the promising performance of the VOLAP prototype implementation suggests that the capabilities of many real-world analytical systems could be enhanced by using these approaches.

1.3.1 Publications


1.4 Organization of Dissertation

This dissertation begins with several chapters that are relevant to the work as a whole, followed by chapters detailing specific aspects of the system which include more specific background information and related work. General background information is presented in Chapter 2. Chapter 3 defines the application model used throughout, as well as the fundamental data types, their representation, and associated algorithms on which the rest of the system is based. Chapter 4 introduces the Hilbert PDC-tree, and includes experiments which test the performance of the tree alone in a non-distributed context. Chapter 5 introduces VOLAP which uses the Hilbert PDC-tree as a primitive to build a fully distributed system which provides similar functionality on much larger scales. A model for investigating query result staleness in the presence of multiple clients is introduced in Chapter 6, which then uses this framework to analyze the behaviour of VOLAP. Finally, Chapter 7 presents a summary of this work, discusses its conclusions, and notes potential areas for future research.

Supplementary material that is relevant but tangential to the core contributions is included in appendices. Appendix A shows the results of some preliminary experiments which guided decisions on data representation, and Appendix B describes the generation of synthetic data and queries for experiments.
Chapter 2

Background

2.1 On-Line Analytical Processing (OLAP)

On-Line Analytical Processing (OLAP) is a widespread technology for knowledge discovery in large database systems. Many essential business applications rely on OLAP for structured data analysis [39].

A defining characteristic of OLAP systems is the use of many hierarchical dimensions to organize data, and the ability to view data with respect to various dimensions at different levels of granularity. Data items in an OLAP system have a multi-dimensional key which identifies the item, and one or more measures which are typically numerical values.

OLAP operations are commonly visualized as transformations from an existing many-dimensional cube into another view with different dimensions. For example, Figure 2.1 shows an OLAP cube for sales with 3 dimensions: time, date, and store. A view with one less dimension can be created by aggregating all values along a particular dimension. For example, aggregating along the store dimension in Figure 2.1 produces a view with two remaining dimensions: date and item.

![Figure 2.1: An OLAP data cube and derived views](image)

6
2.1.1 Dimension Hierarchies

Data in OLAP systems is organized according to several hierarchical dimensions. Dimensions contain discrete values which are not necessarily ordered, like those shown in Figure 2.2. A dimension describes some attribute of the data, and levels in the dimension correspond to different granularities for that attribute. For example, a date dimension could have levels for year, month, and day.

![Hierarchical dimensions for sales from the TPC-DS data set](image)

Data may be viewed from the perspective of any level, where the value for a given level is the aggregate of the results for all its child levels. For example, a query might aggregate all sales of items in a particular year. The result for the given year is the sum of the results for all months within that year, which in turn are the sum of the results for all days within that month.

The ability to efficiently view data at the desired granularity and reduce the number of dimensions by aggregating large fractions of the data is the key functionality of OLAP systems, in contrast to transactional systems which are designed to store and retrieve individual records.

2.1.2 OLAP Operations

OLAP systems provide a set of operations for viewing data with respect to different dimensions of interest at a desired granularity.

The simplest OLAP operation is the slice, which produces a subset of the data by selecting a specific value in some dimension. The result of a slice operation has the same dimensions as the input, but contains only data items with the specified value in the given dimension, as illustrated in Figure 2.3.
The dice operation is similar to the slice operation, but allows specifying any number of values in several dimensions. The result is the subset of the data which matches all the specified values, as shown in Figure 2.4.

The slice and dice operations simply produce subsets of the input data. The more powerful ability to aggregate measures and view data with coarser granularity is provided by the roll-up operation. To perform a roll-up operation, the user specifies a level in one or more dimensions for the result. The system aggregates the values necessary to produce output expressed at the desired level(s). For example, a roll-up operation could be used to view a date dimension in terms of years, rather than quarters, as illustrated in Figure 2.5.

A roll-up operation can also be used to eliminate a dimension entirely by aggregating all values in that dimension. For example, Figure 2.1 shows an OLAP cube with 3 dimensions.
(date, item, and store), and 3 sub-views created by rolling up the corresponding dimension. Rolling up the store dimension produces a view of all sales by date and item, rolling up the date dimension produces a view by store and item, and so on.

Extending this idea, a roll-up operation can be used to aggregate a subset of the data to a single value, like the operation illustrated in Figure 2.6, which aggregates all sales in the year 2016. This is conceptually equivalent to first performing a slice or dice operation to select the relevant data, then performing a roll-up operation on the resulting cube to reduce all dimensions to a single value. A roll-up operation may even aggregate the entire database, which in this example would produce the total sales for all items in all stores for the complete history stored in the database.

From a systems perspective, the demands of OLAP applications are characterized by the need to view data at various levels of granularity, and frequently aggregate very large portions of the database to execute roll-up operations. For example, the transformation illustrated in Figure 2.5 requires aggregating all values for four quarters within a year across the entire database into a single value.

These requirements can lead to performance issues with very large databases that are not carefully designed to suit these demands. For example, a naïve relational schema could degrade to a linear scan of the entire database for many OLAP operations, since no single ordering can provide efficient results for all possible queries.

Existing systems take various approaches to providing adequate performance for OLAP operations, including materializing all cubes (creating separate copies of the data that contain only the aggregated values of interest), extensive indexing, and sophisticated caching mechanisms.
2.2 Spatial Index Trees

Many structures exist for indexing multi-dimensional spatial data. Most tree-based spatial indices can be broadly divided into two categories:

- Space partitioning trees, where each node partitions its space and distributes children accordingly, such as the Quad-tree [31] or KD-tree [10].
- Trees with volumetric keys, which record a bounding region for all children, such as the R-tree [36].

Each category has various advantages and weaknesses, but the latter is more suited to this work for several reasons:

- Trees are balanced and can have high fanout, which results in lower tree height and better space efficiency.
- The tree structure is suitable for storing cached aggregate values for large aggregate queries.
- The structure of the tree itself is independent of the number of dimensions.
- Volumetric trees are better able to handle new or changing data dynamically than space partitioning trees, which often require expensive re-building operations.

A volumetric spatial index tree has a simple structure: each data point or rectangle is stored in a leaf node. All leaf nodes are at the bottom level of the tree. Above the leaf level, leaves are grouped under directory nodes which store a bounding box that encompasses that of all the node’s children. This structure continues upwards recursively, grouping directory nodes until a level has only one node, the root of the tree. The bounding box is typically a simple Minimum Bounding Rectangle (MBR) as in the R-tree.

**Definition 2.1 Minimum Bounding Rectangle (MBR)**

A Minimum Bounding Rectangle, or MBR, is the smallest $d$-dimensional rectangle which contains a set of points. In other words, an MBR is a set of $d$ minimum values and $d$ maximum values, one for each dimension, which describes the range of all coordinates in a set of points.
Directory nodes typically have a high fanout value which is tuned to suit the system architecture. For example, in large disk-backed data structures, fanout is tuned so that directory nodes can have as many children as possible while fitting within a single disk block. Similarly, in-memory structures may be tuned so that nodes fit within a memory page or cache line.

Figure 2.7 illustrates a volumetric spatial index for a set of 8 data points in 2 dimensions. The data set shown on the right consists only of the numbered points, the boxes are shown to illustrate the correspondence with the bounding boxes stored in the tree. However, a similar structure can also be used to index volumetric data, by storing rectangles at the leaves rather than individual points.

The query algorithm for such structures is simple: given a query rectangle \( q \), starting at the root, traverse all children whose bounding boxes intersect \( q \). When any leaf node is encountered that lies within \( q \), include its value in the result.

Most volumetric spatial index trees have similar, and often identical, query algorithms. Variants are typically distinguished by their construction or insertion algorithms, which dictate the organization of the tree and thereby affect query performance.

There are many such variants which describe various bulk construction algorithms, dynamic insertion strategies, auxiliary data, or structural modifications in order to improve performance. In this work, all such data structures that share the general structure of an R-tree are referred to as \textit{R-tree-like}. 
2.3 The Hilbert Curve

The Hilbert curve is a continuous fractal space-filling curve which completely fills the unit square [42]. The \( n \)th approximation of the Hilbert curve, \( H_n \), is a discrete approximation of this curve. Intuitively, it can be viewed as a recursive geometric construction that begins with the first approximation, \( H_1 \), which is duplicated and rotated to form \( H_2 \), and so on, using the same transformation to generate approximations of increasing resolution, as shown in Figure 2.8.

Figure 2.8: The first three iterations of a 2D Hilbert curve construction: \( H_1 \), \( H_2 \), and \( H_3 \).

There are many algorithms for mapping points to and from the Hilbert curve. A detailed description of these approaches is outside the scope of this work, but it is important to note that the idea has been extended to apply to any number of dimensions [16].

A particularly useful property of the Hilbert curve is locality preservation: for any coordinate \((x, y)\) within the unit square, there is a distance \( d \) along the curve to that point, and other such distances close to \( d \) will correspond to coordinates that are relatively close to \((x, y)\). This property makes the Hilbert curve a useful tool for linearizing multi-dimensional coordinates when locality is a concern, such as organizing spatial data in memory or on disk.

The discrete approximations to the Hilbert curve also preserve locality. If the coordinate space is scaled from the unit square to \( 2^n \), then each vertex of the discrete approximation \( H_n \) has integer coordinates. The distance along this curve to any vertex is therefore also an integer, referred to as the Hilbert index. Thus, the discrete approximation \( H_n \) can be used to map multi-dimensional integer coordinates to a single integer index, where \( n \) is dictated by the range of the input coordinates.

**Definition 2.2 Hilbert index**

The Hilbert index of a point is the integer distance along a discrete approximation of the Hilbert curve to that point.
For example, 3-bit coordinates have $2^3 = 8$ possible values along each axis, so $H_3$ can be used to map these coordinates to an integer as illustrated for 2 dimensions in Figure 2.9.

```
    7 | 63 — 62 | 49 — 48 — 47 | 44 — 43 — 42
  ----|--------|----------|----------
    6 | 60 — 61 | 50 — 51 | 46 — 45 | 40 — 41
  ----|--------|----------|----------
    5 | 59 — 56 — 55 | 52 | 33 — 34 | 39 — 38
  ----|--------|----------|----------|----------
    4 | 58 — 57 | 54 — 53 | 32 | 35 — 36 — 37

y

  3 | 5 — 6 | 9 — 10 | 31 | 28 — 27 — 26
  ----|--------|----------|----------|----------
  2 | 4 — 7 — 8 | 11 | 30 — 29 | 24 — 25
  ----|--------|----------|----------|----------
  1 | 3 — 2 | 13 — 12 | 17 — 18 | 23 — 22
  ----|--------|----------|----------|----------
  0 | 0 — 1 | 14 — 15 — 16 | 19 — 20 — 21

  0 1 2 3 4 5 6 7

x
```

Figure 2.9: The third approximation of a 2D Hilbert curve with discrete coordinates

Here, (1, 2) maps to index 7, and indices close to 7 are also close in 2-dimensional space: $6 \Leftrightarrow (1, 3)$, $8 \Leftrightarrow (2, 2)$, $9 \Leftrightarrow (2, 3)$, and so on. All indices from 2 through 9 inclusive map to points that are at most $\sqrt{2}$ distant from (1, 2). Notice, however, that distance is not perfectly preserved: both 13 and 57 are closer to 7 than 10 in 2-dimensional space.

In general, the Hilbert curve preserves locality more strongly from index to point than vice-versa. That is, two nearby Hilbert indices will correspond to two points that are also near one another, but two points that are near one another will not necessarily correspond to nearby Hilbert indices.
Chapter 3

Model and Representation

3.1 Application Model

A real-time OLAP system includes the most recent insertions in query results. The key computational demand on such a system is the ability to aggregate values in the database within some multi-dimensional region at any level in the respective dimension hierarchies.

To facilitate investigation of this aspect of the system, a simplified real-time OLAP application model is used, where applications interact with the system via two fundamental operations: insert and aggregate query. These operations can be used directly for many analytical tasks, but could also serve as primitives for higher level operations in an OLAP system with more features. For example, a production OLAP system may require operations that are not defined in this model, such as materializing multi-dimensional reports or views like those illustrated in Section 2.1. A roll-up of a dimension to produce a view with fewer dimensions would execute several aggregate queries to calculate the values to be used in the new view. The software design and engineering issues surrounding such functionality are outside the scope of this work, which focuses on the performance-critical core of such a system which is characterized by these operations.

In this model, the system represents a database with \( N_i \) data items and \( d \) hierarchical dimensions \( D_1, D_2, \ldots, D_d \). A data item is identified by a \( d \)-dimensional \textit{point}, and contains one or more \textit{measure} values of interest.

**Definition 3.1 Point**

A \textit{point} specifies \( d \) values, one for each dimension in the database, at the finest (leaf) level of each dimension.
Definition 3.2 Measure
A measure is a typically numerical value of interest which does not reside within a
dimension hierarchy, and can be combined with another measure by some associative
and commutative binary operation.

Definition 3.3 Data item
A data item is the combination of a point key, which identifies the item, and one or more
measure values.

For example, using the dimension hierarchies shown in Figure 2.2, a data item for the
sale of a $100 Acme tool in Ottawa on December 1st at noon can be represented in tabular
form as follows:

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Measures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item</td>
<td>Store</td>
</tr>
<tr>
<td>Home.Tools.Acme</td>
<td>Canada.Ontario.Ottawa</td>
</tr>
</tbody>
</table>

3.1.1 Operations
The user interacts with the system by sending an ordered stream of inserts \( I_i \) and aggregate
queries \( Q_j \), and receiving an asynchronous stream of corresponding acknowledgements \( R_i \)
and query results \( R_j \), as shown in Figure 3.1. Requests (inserts or aggregate queries) specify
an integer identifier which is included in the corresponding response to allow clients to
associate responses with their originating requests.

![Figure 3.1: Real-time OLAP application model](image)

Figure 3.1: Real-time OLAP application model
Definition 3.4 Insert
An insert specifies a multi-dimensional point and a set of measure values, which together describe a new data item to be inserted into the database. The response to an insert evaluation, or insertion, is either an acknowledgement or an error notification.

Queries request the aggregate value of all measures within a specific region. The abstract notion of a box is used for any description of a region within the dimension hierarchies which may or may not be contiguous. Boxes are an abstract concept to account for different representations that serve the same purpose but have different performance characteristics.

Definition 3.5 Box
A box is a d-dimensional region or set of non-contiguous regions within the dimension hierarchies. A bounding box is a box that encloses a set of points or other boxes.

Boxes provide a way to describe an arbitrary region in the multi-dimensional key space of the database, which, along with an aggregation function, can be used to specify a query which aggregates all measures within a given region.

Definition 3.6 Aggregate query
An aggregate query, or simply query, specifies an aggregation function A and a multi-dimensional bounding box which specifies the subset of the data to be aggregated using A. The result of a query is the aggregate value of all measures within the specified region, or an error.

When discussing queries, it is useful to have a metric against which to measure expected performance and other factors. Coverage in terms of data item count provides such a metric which is useful for evaluating performance, simple to measure in practice, and oblivious to the structure of the dimension hierarchies or distribution of data within the database.

Definition 3.7 Coverage
The coverage of a query Q is the percentage of the data contained in the region given by Q, that is, the percentage of data items included in the aggregate result.
The system also provides other basic operations, such as the ability to request the current extent and count of all items in the database, but since such operations are not of theoretical interest, they are omitted here for brevity.

3.1.2 Consistency

In a sequential context, this would conclude the definition of the system’s interface, but when considering parallel and asynchronous execution, a consistency model must be provided to define the guarantees that the user may depend on. Two such models are of interest here: serializable execution and eventual consistency.

The strongest guarantee is a serializable execution, which is effectively identical to execution in a serial context.

**Definition 3.8 Serializable**

A serializable [12] execution of an operation stream has the same user-visible effect as a serial (i.e. non-parallel) execution of that stream.

Note that only user-visible effects are relevant to the definition of serializable execution, operations are not required to be executed serially inside the system. In this model, the relevant operations have properties which allow for a more flexible definition of serializable execution. In particular, insertions are write-only and only affect the outcome of later queries, so any order of execution for a series of successive insertions is indistinguishable from another by the user. Similarly, queries are read-only and do not affect the result of other operations, so any order of execution for a series of successive queries is also indistinguishable from another by the user. Therefore, in this model, a serializable execution can be equivalently defined as a guarantee that each query contains all relevant results from insertions issued earlier in the stream, and none of those issued later.

Serializable execution is a strong guarantee that only applies when the input stream has a meaningful order. This is the case for an individual client’s operation stream, but does not apply globally in a distributed system with multiple clients. For example, if the system is being used by several clients that are geographically separated, their operation streams are distinct and subject to different network routing and latency. In such cases, there is no single objective order for operations that can be applied to both streams. Though the
operations will arrive at the server in some order, this order is not meaningful to clients
due to non-deterministic network routing and other factors. That is, if one client sends an
operation “earlier” than another, it is not guaranteed that the server receives the operations
in the same order.

Because of this, serializable execution across several client streams based on the order in
which operations are received by the server does not provide any useful guarantees to clients.
A weaker consistency model with less performance impact, like eventual consistency is
therefore more suitable in this context.

**Definition 3.9** Eventual consistency

*Eventual consistency*, for a stream of insertion and queries, guarantees that an insertion
will eventually be seen by subsequent queries, and if no further insertions occur within
a given box, eventually all subsequent queries will return equivalent results for that box.

Non-serializable execution implies that queries may fail to include data from insertions
issued at an earlier point in time from another client. *Staleness* is a measure for how
out-of-date query results are in such scenarios.

**Definition 3.10** Staleness

*Staleness* is the elapsed time from the most recent insertion considered in a query result
to the time of the query. For example, if the result of a query issued at time $t$ has a
staleness of 1 second, it includes all relevant data from insertions issued before time $t - 1$ seconds.

Though eventual consistency is a very weak theoretical guarantee, it is commonly used
in systems where providing strong order guarantees would be prohibitively expensive, and
systems that only guarantee eventual consistency can provide good bounds on staleness in
practice [71].

The key distinction between serializable execution and eventual consistency is that
the former provides a strong correctness guarantee that is independent of time, while the
latter provides a weak guarantee that relies on empirical methods to determine the expected
staleness of query results.

In other words, the staleness of a serializable execution is always zero, while the staleness
in eventual consistency is unbounded, but can be measured empirically and predicted using statistical methods. Such a method for analyzing and predicting staleness in VOLAP when serializable execution is not available is described in detail in Chapter 6.

### 3.2 Hierarchical Identifiers (IDs)

A value in a dimension is referred to by a hierarchical ID. An ID is a sequence of indices in different levels within a dimension.

<table>
<thead>
<tr>
<th><strong>Definition 3.11 Level</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>A level is the depth of value within a dimension hierarchy, where the top (coarsest) value in a dimension has level 1. The level of an ID ( a ) is denoted ( \text{lev}_a ).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Definition 3.12 ID</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>An ID is a sequence of indices which refers to a value at some level within a single dimension. An ID is denoted ( d.i.j.k \ldots ) where ( d ) is the dimension number and ( i, j, k, \ldots ) are the indices within the corresponding levels. For example, ( 1.4.3.17 ) is an ID in dimension 1, with index 4 in level 1, and so on. The special notation ( d.0\ldots ) refers to the top level ID in dimension ( d ).</td>
</tr>
</tbody>
</table>

The most coarse, or top, level in a dimension is level 1, its child is level 2, and so on. Note that the notion of “higher” or “lower” levels is the inverse of numerical order, similar to tree depth, that is:

\[
    l < m \iff \text{level } l \text{ is higher than level } m
\]

\[
    l > m \iff \text{level } l \text{ is lower than level } m
\]

Similarly, to raise an ID is to create an ancestor ID at a higher level. Importantly, the ID representation is such that any ancestor of an ID shares a common prefix with that ID.

<table>
<thead>
<tr>
<th><strong>Definition 3.13 Ancestor</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>The level ( l ) ancestor of an ID ( a ), denoted ( \text{anc}_l a ), is ( a ) raised to level ( l ), and is only defined when ( \text{lev } a \geq l ). For example, ( \text{anc}_2 2.15.6.3 = 2.15 ).</td>
</tr>
</tbody>
</table>
IDs within a dimension have a partial ordering, \textit{subsumes}, which accounts for hierarchies and allows new indices to be allocated at a given level without imposing constraints on the indices available for values at other levels. Comparisons of spatial regions are defined in terms of subsumption to avoid imposing a total ordering which would give more preference to some values over others and restrict extension of the dimension hierarchies [29].

\begin{definition}
\textbf{Subsumes}
Given two IDs \(a\) and \(b\) within the same dimension, \(a\) \textit{subsumes} \(b\), or \(a \succeq b\), if and only if \(a = b\) or \(a\) is a direct or indirect ancestor of \(b\). An ID never subsumes an ID in a different dimension.
\end{definition}

\subsection{Representation}
IDs are implemented as integers divided into segments to represent the dimension and index within each \textit{level} of the dimension hierarchy, as shown in Figure 3.2.

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Dimension & Level 1 & Level 2 & Level 3 & \ldots \\
\hline
\end{tabular}
\caption{Hierarchical ID structure}
\end{figure}

This self-contained structure allows IDs to be viewed at any level by simply masking bits. The number of bits used for each segment of an ID as well as the total number of levels may be chosen to suit a given system or data set. The appropriate values depend on the number of dimensions and the breadth of each level in the data set.

The key advantage of this self-contained representation is that determining whether one ID subsumes another can be done very quickly without consulting any auxiliary data structures. With this representation, subsumption can be determined using only basic integer operations, since

\[ a \succeq b \iff (a = b) \lor (\text{lev} a \leq \text{lev} b \land a = b \land M_{\text{lev} a}) \]

where \(M_l\) is a mask where the leftmost bits up to and including level \(l\) are set to 1, and “&” is the bitwise AND operator. For example, to compare \(a = 1.4.3\) with \(b = 1.4.5.96\):
\[
a \succeq b \iff (a = b) \lor (\text{lev } a \leq \text{lev } b \land a = b \& M_{\text{lev } a})
\]
\[
\iff (1.4.3 = 1.4.5.96) \lor (3 \leq 4 \land 1.4.3 = 1.4.5.96 \& M_3)
\]
\[
\iff 1.4.3 = 1.4.5.96 \& M_3
\]
\[
\iff 1.4.3 = 1.4.5
\]
\[
\therefore a \not\succeq b
\]

Subsumption defines a canonical partial ordering for IDs which is distinct from the total ordering of the integer representation of IDs. For example, above, \( a \not\succeq b \) even though \( a < b \) in representation\(^1\). However, the two are related in ways that prove useful for the storage and retrieval of IDs:

**Theorem 3.1 ID subsumption ordering**

For any IDs \( a \) and \( b \), \( a > b \Rightarrow a \not\succeq b \). That is, if the integer representation of \( a \) is greater than that of \( b \), then \( a \) does not subsume \( b \).

**Proof.** Since \( a > b \), the definition of subsumption can be simplified:

\[
a \succeq b \iff (a = b) \lor (\text{lev } a \leq \text{lev } b \land a = b \& M_{\text{lev } a})
\]
\[
\iff \text{lev } a \leq \text{lev } b \land a = b \& M_{\text{lev } a}
\]

Since \( a \neq b \)

Then, there are three cases to consider:

1. If \( a \) is on a lower level than \( b \), then \( \text{lev } a > \text{lev } b \) trivially implies \( a \not\succeq b \).

2. If \( a \) and \( b \) are on the same level, then:

\[
a \succeq b \iff a = b \& M_{\text{lev } b}
\]
\[
\iff a = b
\]

Since \( \text{lev } a = \text{lev } b \) and \( b \& M_{\text{lev } b} = b \)

\[
\therefore a \not\succeq b
\]

Since \( a > b \)

---

\(^1\) The notation \( a \succeq b \) is chosen to be analogous to the set notation \( a \supseteq b \), though in representation subsumption is more closely related to \( a < b \).
3. Finally, if \( a \) is on a higher level than \( b \), then:

\[
a \preceq b \iff a = b \land M_{lev\,a} \\
\iff a \leq b \\
\therefore a \not\preceq b
\]

Since \( lev\,a < lev\,b \)

\[
\iff b \land M_{lev\,a} \leq b
\]

Since \( a > b \)

Note that \( b \land M_{lev\,a} \) sets some number of least-significant bits in \( b \) to 0. Therefore, for \( a = b \land M_{lev\,a} \) to be true, it must be possible to make \( b = a \) by setting bits in \( b \) to 0. This is clearly impossible, since \( a > b \) and zeroing bits can only make \( b \) smaller.

Therefore, in all cases where \( a > b \), \( a \not\preceq b \).

\[\square\]

### 3.2.2 Operations

Most ID operations such as retrieving the dimension number are straightforward to implement with basic integer operations. Given the configured number of bits allocated to each level, the system can calculate and store the necessary values and masks in small lookup tables for efficiency. For example, if \( M \) is an array of masks indexed by level where elements have all bits in the corresponding level and above set, then the level 2 ancestor of an ID \( a \) is simply \( a \land M[2] \), assuming \( a \) is at a lower level.

Given the relevant levels, the necessary operations can be implemented in a similarly simple and efficient constant-time fashion. However, this highlights a difficulty with the ID representation which does not store an explicit level. For example, the parent of \( a \) is \( a \land M[lev(a) - 1] \), but in order to calculate this value, \( lev\,a \) must be known. Though the level of IDs is often known from context, this is not always the case.

Given the limited space available, storing the level explicitly would significantly reduce the number of IDs that can be represented, and doing so would make many operations more expensive due to the need to maintain the level number whenever an ID is modified. Though calculating the level of an ID can be done with a loop that simply compares against a mask for every level, the introduction of control flow and branching instructions in these low level operations would significantly impact performance.

Fortunately, modern hardware architectures provide facilities which can be used to solve this problem efficiently without branching. The find first set or FFS instruction returns
the position of the least significant bit in an integer that is set. Given this, it remains to determine which level the rightmost set bit falls within. This can be done efficiently using a lookup table indexed by bit position which contains the corresponding level numbers, which is straightforward to calculate given the configured number of bits used for each level. For example, for 4-level 32-bit IDs that allocate 4 bits to each of the top 3 levels, and the remaining 20 bits to the lowest level:

\[ L = [1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4] \]

Then, the level of an ID \( a \) is simply \( L[\text{FFS}(a)] \).

### 3.3 Minimum Describing Sets (MDSs)

A common simple way to describe a multi-dimensional area is with a Minimum Bounding Rectangle, or \( MBR \). An MBR is a multi-dimensional rectangle that encloses a set of points. MBRs are limited to describing a contiguous range in each dimension, which can result in a much larger area being covered than is necessary to enclose a given set of points.

A Minimum Describing Set (MDS) can be used to mitigate this problem. Rather than describing a single contiguous range for each dimension, an MDS describes a set of values. Figure 3.3 illustrates the difference between an MBR and MDS that enclose the same points. The region described by an MDS is usually smaller, and never larger, than the region described by an MBR that encloses the same points, though the description of the MDS itself may be larger since it lists individual values rather than ranges.

Using MDSs to describe regions allows a system to be less susceptible to overlap, particularly as the number of dimensions increases. For example, consider the regions shown in Figure 3.3 as bounding boxes in a data structure. If a value with \( Y = 1000 \) were inserted into this subtree, the MBR’s volume would increase dramatically to cover the entire range from \( Y = 1 \ldots 1000 \). However, the MDS would only add a thin slice for \( Y = 1000 \). Using the MBR, a query for \( Y = 500 \) would deduce that points may exist within the query region, but the MDS correctly shows that no such points exist.

---

2. Some architectures instead provide similar instructions, such as \textit{count trailing zeroes} which can be efficiently used for the same purpose.
Definition 3.15 Minimum Describing Set (MDS)
Given $d$ dimensions $D_i$, let $S$ be a set of points, that is, a set with elements of the form $(d_1, d_2, \ldots, d_d) : d_i \in D_i$, and let $l_i$ be a relevant level for each dimension $D_i$.
A Minimum Describing Set for $S$ is a sequence $(M_1, M_2, \ldots, M_d) : M_i \subseteq D_i$ which satisfies three conditions for all $i$ where $1 \leq i \leq d$:

1. **Coverage**: $\forall (d_1, d_2, \ldots, d_d) \in S$, $\exists m \in M_i$ where $m \succeq d_i$.

2. **Comparability**: $\forall m \in M_i$, $\text{lev} m = l_i$.

3. **Minimality**: $\forall (N_1, N_2, \ldots, N_d) : N_i \subseteq D_i$ which satisfy the above, $M_i \subseteq N_i$.

Several useful relationships between MDSs can be defined that are analogous to their rectangular counterparts, such as whether or not one MDS contains another:

**Definition 3.16 Contains**
An MDS $N$ contains $M$ if and only if $\forall m_i \in M$, $\exists n_i \in N$ where $n_i \succeq m_i$. That is, every ID in $M$ is subsumed by some ID in $N$.

A more fine-grained description of the overlap between MDSs is given by the coverage, which is particularly useful for executing queries:
Definition 3.17 Coverage
The coverage of an MDS $N$ over another MDS $M$ has one of three possible values:

1. **Full**: $N$ contains $M$.

2. **Partial**: $\forall i : 1 \leq i \leq d, \exists n \in N_i, m \in M_i : n \succeq m$. That is, there is at least some overlap in each dimension.

3. **None**: $\exists i : \nexists n \in N_i, m \in M_i : n \succeq m$. That is, there is no overlap in at least one dimension, so the geometric intersection of $N$ and $M$ is empty.

3.3.1 Representation

Theorem 3.1 shows a useful relation between ID subsumption and integer comparison. The utility of this relationship for storing sets of IDs is better illustrated by its contrapositive: for any IDs $a$ and $b$, $a \succeq b \Rightarrow a \leq b$. That is, all $a$ which subsume $b$ have an integer representation not greater than that of $b$.

This suggests a straightforward representation for sets of IDs: simply store IDs as integers in a sorted array. Since IDs are fully self-contained, this representation is sufficient to represent MDSs with implicit dimension boundaries and levels, as shown in Figure 3.4.

![Figure 3.4: Flat MDS representation](image)

This representation is suited to the most important MDS algorithms, since they can be implemented as linear scans over contiguous chunks of memory as shown in the following section. Though a more complex representation could achieve better asymptotic bounds for some operations, MDSs are typically small, and the ideal cache behaviour of linear scanning yields good performance in practice. Experimental investigations detailed in Appendix A have shown that the benefits of a compact contiguous representation are dominant in larger systems.
3.3.2 Operations

Primitives

The MDS algorithms used to execute queries are described here in terms of the simple abstract iterator interface shown in Table 3.1, with some details and optimizations omitted for brevity.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Returns</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin($M$)</td>
<td>An iterator to the first ID in $M$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>end($M$)</td>
<td>An iterator one past the last ID in $M$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>next(i)</td>
<td>An iterator one past $i$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>i</td>
<td>The ID pointed to by $i$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>next_l(i)</td>
<td>The first iterator $j$ past $i$ that is distinct from $i$ at level $l$</td>
<td>$O(j - i)$</td>
</tr>
<tr>
<td>scan(i, j, a)</td>
<td>The first ID in range $[i, j]$ that is $\geq a$</td>
<td>$O(j - i)$</td>
</tr>
</tbody>
</table>

Table 3.1: MDS iterator primitives

The next_l operation is used to “view” an MDS in-place as if it were expressed at the higher level $l$. For example, iterating over the MDS $\{1.2.1, 1.2.2, 1.3.5, 1.6.2, 1.6.7\}$ with next_2 would produce the sequence $(1.2, 1.3, 1.6)$.

The scan operation searches forward from a given position to find the first ID which is not less than a given value. This is used to skip to the next ID of interest when comparing MDSs, which simplifies the description of the algorithms themselves.

Containment

Algorithm 3.1, mds_contains, determines whether or not one MDS contains another (Definition 3.16).

**Lemma 3.2** MDS contains time complexity

Algorithm 3.1, mds_contains, runs in $O(n + m)$ time, where $n$ and $m$ are the sizes of the input MDSs $Q$ and $B$.

*Proof.* The loop in the algorithm runs at most once for every element in $B$, making $|B| = O(m)$ steps, each of which performs a constant amount of work except for advancing the $q$ iterator. Though $q$ may be advanced many times in a single loop iteration, it can only be
Algorithm 3.1: mds.contains

**Input:** \( Q \), the query MDS  
**Input:** \( B \), the box MDS to test  
**Result:** True if and only if \( Q \) contains \( B \)

1. Let \( q = \text{begin}(Q) \)
2. Let \( b = \text{begin}(B) \)
3. while \( b \neq \text{end}(B) \) do
   4. Let \( d = \text{dim}(b) \)
   5. if \( \text{dim}(q) \neq d \) then
      6. \( q := \text{scan}(q, \text{end}(Q), d.0...) \) \hspace{1cm} (Scan to start of next dimension)
   7. Let \( l = \text{lev}(q) \)
   8. if \( \text{lev}(b) \leq l \) then
      9. return false \hspace{1cm} (Can’t contain IDs at a higher level)
   10. Let \( \hat{b} = \text{anc}_l(b) \)
   11. \( q := \text{scan}(q, \text{end}(Q), \hat{b}) \)
   12. if \( q = \text{end}(Q) \lor q \neq \hat{b} \) then
      13. return false \hspace{1cm} (No ID found that subsumes \( \hat{b} \))
   14. \( b := \text{next}(b) \)
15. return true

advanced at most \( |Q| = O(n) \) times, for a total of \( O(n) \) time amortized over all loop iterations. Therefore, the total running time of the algorithm is \( O(n) + O(m) = O(n+m) \). \( \square \)

Coverage

Algorithm 3.2, mds.coverage, determines the coverage of one MDS over another (Definition 3.7). This algorithm is slightly more complicated since it supports comparing MDSs at any level and may therefore need to “view” either MDS at a higher level, unlike mds.contains which simply terminates if the query MDS is found to be at a lower level than the other.

The progress of mds.coverage where the query MDS is at a higher level is illustrated in Figure 3.5. Arrows indicate iterator steps, dotted lines show ID comparisons, with thick dotted lines indicating matches. In this example, the result is partial coverage, since \( Q \) does not cover 1.7, 2.9.3, or 2.9.6.
Lemma 3.3 MDS coverage time complexity

Algorithm 3.2, mds_coverage, runs in $O(n + m)$ time, where $n$ and $m$ are the sizes of the input MDSs $Q$ and $B$.

Proof. Though the control flow in this algorithm is more complicated than in mds_contains, a similar argument to Lemma 3.2 applies: the two inputs are scanned in tandem, where at least one iterator is advanced every step of the algorithm, and a constant amount of work is performed for every step.

Note that these algorithms are variants of the classical algorithms for finding the union or intersection of sets stored in sorted arrays [51, §5.2.4]. This is particularly visible in lines 14 through 21 in mds_coverage. They have been modified to handle hierarchical dimensions, and track the degree of coverage rather than produce a result set, but the progress of iteration is identical so similar analytical arguments apply.
Algorithm 3.2: mds_coverage

Input: Q, The query MDS
Input: B, The box MDS to measure coverage of
Result: FULL, PARTIAL, or NONE

1. Let full = true

2. Let $q = \text{begin}(Q)$

3. Let $b = \text{begin}(B)$

for $d = 1 \ldots D$ do

4. Let some = false

if $\text{dim}(\hat{q}) \neq d$ then

5. $q := \text{scan}(q, \text{end}(Q), d.0\ldots)$

6. (Move to dimension start in Q)

if $\text{dim}(\hat{b}) \neq d$ then

7. $b := \text{scan}(b, \text{end}(B), d.0\ldots)$

8. (Move to dimension start in B)

while $\text{dim}(\hat{q}) = d \land \text{dim}(\hat{b}) = d$ do

9. Let $l = \min(\text{lev}(\hat{q}), \text{lev}(\hat{b}))$

10. Let $\hat{q} = \text{anc}_l \hat{q}$

11. Let $\hat{b} = \text{anc}_l \hat{b}$

12. if $\hat{q} < \hat{b}$ then

13. $q := \text{next}_l(q)$

14. (Advance q)

15. else if $\hat{b} < \hat{q}$ then

16. $b := \text{next}_l(b)$

17. full := false

18. (b is not covered)

19. else

20. $q := \text{next}_l(q)$

21. $b := \text{next}_l(b)$

22. some := true

23. if $\neg\text{full}$ then

24. break

(No need to further search in this dimension)

25. if $\neg\text{some}$ then

26. return NONE

27. if $q = \text{end}_d(Q) \land b \neq \text{end}_d(B)$ then

28. full := false

(Reached end of dimension in Q without covering $\hat{b}$)

29. return FULL if full = true, otherwise PARTIAL
3.4 Summary of Common Notation

Table 3.2 shows a summary of the notation introduced in this chapter which is used consistently throughout the remainder of this work.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$</td>
<td>Number of dimensions</td>
<td>Section 3.1</td>
</tr>
<tr>
<td>$N_i$</td>
<td>Total number of data items</td>
<td>Section 3.1</td>
</tr>
<tr>
<td>$I_x$</td>
<td>Insert operation $x$</td>
<td>Definition 3.4</td>
</tr>
<tr>
<td>$Q_x$</td>
<td>Query operation $x$</td>
<td>Definition 3.6</td>
</tr>
<tr>
<td>$R_x$</td>
<td>Response to / result of $I_x$ / $Q_x$</td>
<td>Definitions 3.4, 3.6</td>
</tr>
<tr>
<td>lev $a$</td>
<td>Level of ID $a$</td>
<td>Definition 3.11</td>
</tr>
<tr>
<td>anc$_l$ $a$</td>
<td>The ancestor of ID $a$ at level $l$</td>
<td>Definition 3.13</td>
</tr>
<tr>
<td>$a \succeq b$</td>
<td>$a$ subsumes $b$ ($a = b \lor a$ is an ancestor of $b$)</td>
<td>Definition 3.14</td>
</tr>
<tr>
<td>$d.0\ldots$</td>
<td>The top level ID in dimension $d$</td>
<td>Definition 3.12</td>
</tr>
</tbody>
</table>

Table 3.2: Summary of common notation
Chapter 4
The Hilbert PDC-tree

4.1 Introduction

R-tree-like data structures are commonly used for indexing multi-dimensional data, and perform well for suitable problems and data sets. However, their performance rapidly degrades when used with a very large number of dimensions. This effect is primarily due to overlap between the bounding boxes for tree nodes, which increasingly becomes an issue as the number of dimensions increases. When the overlap between sibling nodes is high, it is more likely that several subtrees will need to be traversed when calculating the result for an aggregate query. With a high enough number of dimensions, even queries for relatively small regions may need to traverse the majority of the tree. The overhead of geometric calculations during insertion also becomes increasingly problematic as the data size and number of dimensions increases.

Additionally, R-tree-like structures are not designed to support the hierarchical dimensions required by OLAP applications. Several data structures have been proposed which address some of these issues, but often at significant cost. In particular, insertion performance is often sacrificed to improve query performance or add support for hierarchical dimensions.

The Hilbert PDC-tree is designed to address these issues while supporting a much higher rate of data ingestion compared to previous DC-tree variants. Specifically, the Hilbert PDC-tree supports hierarchical dimensions, scales well to a large number of dimensions, and supports parallel execution of many insert and query operations. The Hilbert curve is used to determine the position for a new element, reducing the amount of geometric computation that must be performed during insertion considerably. The locality-preserving nature of the Hilbert curve, along with a novel overlap-minimizing split algorithm, maintains good query performance despite a significant increase in ingestion throughput.

---

This chapter is a more detailed description of previously published work [65].

31
4.2 Related Work

The R-tree [36] is the classical data structure for indexing multi-dimensional data sets. Over the years, numerous data structures based on the R-tree have been introduced.

Many such structures are refinements to the insertion algorithm which improve query performance at the cost of insertion performance. Among the most well-known of these is the R*-tree [7][8], which occasionally re-inserts records to maintain a better structure for querying. Other approaches change the tree representation itself. The CR-tree [50] uses a compressed MBR representation to pack more information in a single tree node, for more efficient use of memory.

An appropriate R-tree-like structure is an effective choice for indexing data sets with few dimensions, but all suffer from performance degradation when the number of dimensions gets too large. The X-tree [11] improves this situation somewhat by using an overlap-minimizing split algorithm, and introducing supernodes which have a higher fanout than usual if a good split can not be found. However, the performance problems of R-trees with many dimensions are inherent to the use of MBRs in a tree structure.

To overcome this problem, the DC-tree [29] introduced the use of Minimum Describing Sets (MDSs), which can describe many non-overlapping regions in a single key. Since the values in a dimension in an MDS are not necessarily contiguous, the DC-tree is much less susceptible to overlap as the number of dimensions increases. As a result, query performance scales more gracefully as the number of dimensions increases compared to MBR-based trees. Aside from performance, the DC-tree has characteristics that are particularly suitable for OLAP systems: support for hierarchical dimensions, and fast aggregation of large fractions of the data.

The PDC-tree [27] improves overall performance in a multi-core environment, by allowing the tree to be safely used by many threads at once. A minimal locking scheme ensures that threads are only blocked for a short period of time, allowing good speedup to be achieved by using more threads. Other structures for linearly ordered data have used a similar approach [56].

The downside of DC-trees is that working with MDSs is more expensive than MBRs. As with R-trees, many geometric calculations must be performed when modifying the tree, which can be a bottleneck on insertion performance. This problem has been addressed
for R-trees: the Hilbert R-tree [48] avoids these calculations by instead inserting records according to the Hilbert index of their keys. Reducing keys to a linear ordering results in a significantly faster insertion algorithm.

The Hilbert PDC-tree is essentially a merging of the PDC-tree with the insertion strategy of the Hilbert R-tree, which retains the features of the PDC-tree and improves insertion performance by using the Hilbert curve to organize data. The remainder of this section describes the details of these structures which are relevant to the design of the Hilbert PDC-tree.

4.2.1 The DC-tree

The DC-tree has a structure similar to an R-tree, but uses MDSs for bounding boxes rather than MBRs. Its insertion and split algorithms are designed to work with hierarchical data and minimize overlap. The ID and MDS representation used in DC-tree, however, differ significantly from those described in the previous chapter which are used in the Hilbert PDC-tree.

The DC-tree uses 32-bit IDs where the level is stored in the most significant bits, and the remaining space is used for an index within that level of the dimension, as shown in Figure 4.1. The dimension of an ID is known implicitly by storing each dimension in an MDS separately. Auxiliary dictionaries store the parent of each value in the dimension hierarchy, allowing an MDS to be raised to higher levels at the cost of looking up values in auxiliary data structures and constructing a new MDS.

<table>
<thead>
<tr>
<th>Level</th>
<th>Index within level</th>
</tr>
</thead>
</table>

Figure 4.1: IDs in the DC-tree

Though the details differ, at a high level the DC-tree operates much like an R-tree. Insertion location and the distribution of children during node splits is determined by geometric calculations such as volume, expansion, and overlap. Aggregate queries descend from the root, visiting any children that intersect the query region.

To support fast aggregation of large fractions of the data, directory nodes in the DC-tree store a cached aggregate of all values in the subtree rooted at that node, as shown in Figure 4.2. Thus, if the node’s key is completely contained by the query region, there is no need to continue the search to any children of that node.
Figure 4.2: A DC-tree for a set of 8 points. In the data, areas covered by an MDS are shown shaded, and rectangular MDS bounds are shown as dashed outlines. The MDS of the root node is not shown for clarity. Note the contrast in covered area with the R-tree shown in Figure 2.7.

### 4.2.2 The PDC-tree

The PDC-tree is a parallel evolution of the DC-tree which uses a minimal locking scheme to allow several threads to manipulate and query the tree at once. The fundamental algorithms are similar to those of the DC-tree, but allow for speedup to be achieved in a multi-core environment.

The PDC-tree adds a lock and time stamp to each directory node in the tree, but ensures that locks are only held briefly to manipulate nodes, and never to lock an entire subtree. Each insertion atomically increments a monotonic counter, and uses this value to set a time stamp on nodes which are created when a full node is split. The query algorithm uses this information to determine if a node has been split during query execution, and uses horizontal links to update the query result as necessary.

Thus, the PDC-tree provides a multi-threaded consistency model based on time: a query result is guaranteed to include any insertions which arrived at the tree earlier than the query, where “earlier” is defined by the order in which threads increment the monotonically increasing time stamp.

The PDC-tree improves performance in a multi-threaded environment, but still suffers from the overhead of working with MDSs, particularly during data ingestion where a large number of geometric calculations are performed.
4.2.3 The Hilbert R-tree

The Hilbert R-tree dispenses with the insertion algorithm of the R-tree, and instead inserts new items in a position determined by a Hilbert mapping. To insert a new item, the key is first mapped to an integer Hilbert index using a discrete approximation of the Hilbert curve.\footnote{1. See Section 2.3 for details.}

This index is then used to insert the new item as it would be in a linear tree using the index as the key. The insertion algorithm is essentially equivalent to that of a B+-tree\footnote{21}, but the MBR of nodes is maintained for use by queries. The structure of the tree is thus very similar to an R-tree, but nodes also store the maximum Hilbert index of all children nodes which is used to determine insert positions.

The query algorithm of the Hilbert R-tree is equivalent to that of the R-tree, and does not make use of the stored Hilbert indices directly.

4.3 Structure of the Hilbert PDC-tree

The basic structure of the Hilbert PDC-tree is similar to most volumetric spatial index trees: leaves contain the key and measure values for a data item, and directory nodes contain a bounding box key which encompasses the keys of all its children. For these bounding boxes, the Hilbert PDC-tree uses the self-contained ID and MDS representations described in Chapter 3. Note that these MDSs are conceptually similar to those used in the DC-tree, but importantly use a new representation which eliminates the need for auxiliary data structures.

Like the DC-tree, each directory node contains the cached aggregate value of all the measures in the subtree rooted at that node.

Like the X-tree, directory nodes may be supernodes with a larger fanout than usual.

Like the Hilbert R-tree, directory nodes contain the maximum Hilbert index of all the children of that node which is used during insertion.

Like the PDC-tree, each directory node contains a lock which is briefly held when necessary. However, the Hilbert PDC-tree provides a different parallel consistency guarantee than the PDC-tree, and uses a slightly different locking strategy. Directory nodes that have been split also contain pointers to their left and right immediate \textit{split descendants}. 
**Definition 4.1 Split descendant, split ancestor**

A split descendant $b$ of a node $a$ is a node that was created as the result of $a$ being split, or any split descendant of $a$ being split. If $b$ was directly created as the result of splitting $a$, then $b$ is an immediate split descendant of $a$.

If $b$ is a split descendant of $a$, then $a$ is referred to as a split ancestor of $b$.

The structure of the Hilbert PDC-tree is illustrated in Figure 4.3.

![Figure 4.3: A Hilbert PDC-tree for a set of 8 points. Hilbert indices, shown in italic, are calculated for a 4-bit coordinate space.](image)

### 4.4 Notation

The primitive node operations used in algorithms are shown in Table 4.1.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>root()</td>
<td>Atomically returns the locked root node of the tree</td>
</tr>
<tr>
<td>hindex($n$)</td>
<td>Returns the Hilbert index of node $n$</td>
</tr>
<tr>
<td>parent($n$)</td>
<td>Returns the parent pointer of node $n$</td>
</tr>
<tr>
<td>children($n$)</td>
<td>Returns the child array of node $n$</td>
</tr>
<tr>
<td>descendants($n$)</td>
<td>Returns an array of the “sides” created if $n$ has been split</td>
</tr>
<tr>
<td>lock($n$)</td>
<td>Locks node $n$</td>
</tr>
<tr>
<td>unlock($n$)</td>
<td>Unlocks node $n$</td>
</tr>
<tr>
<td>unlock($S$)</td>
<td>Unlocks all nodes in array $S$</td>
</tr>
</tbody>
</table>

Table 4.1: Tree node primitives

Where possible, scoped notations that make the extent of critical regions clear are used rather than explicit lock and unlock operations. The behaviour of these syntactic constructs is described in Table 4.2.
### Notation and Behaviour

**with locked** $x = \ldots$ **do**<br>Locks $x$ for the extent of the following scope.<br>Guarantees $x$ is unlocked at scope end or on return.<br>**foreach locked** $x \in \ldots$ **do**<br>A **foreach** loop which locks each element.<br>Each element is locked for one loop iteration.

**Table 4.2:** Scoped lock notation

### 4.5 Insertion

The Hilbert PDC-tree uses the Hilbert index of keys rather than geometric comparison to quickly determine an insert position. This is conceptually similar to the approach used in the Hilbert R-tree [48], but the hierarchical nature of MDS keys presents new challenges. Additionally, the simplistic split algorithm of the Hilbert R-tree often produces an unnecessarily high amount of overlap, which is problematic for OLAP applications with many dimensions.

This section describes how the Hilbert PDC-tree adapts this approach to hierarchical data, as well as a new split algorithm which minimizes overlap within the constraints imposed by the linear Hilbert ordering.

#### 4.5.1 Mapping Points to a Hilbert Index

The Hilbert curve can be used to map arbitrary multi-dimensional points to a one-dimensional index, but does not account for hierarchical values. In particular, MDSs in the tree are expressed at various levels, where nodes higher in the tree are likely to have keys at higher levels in the dimension hierarchy. Consequently, MDSs are often compared at different levels than the leaf levels for which the Hilbert indices stored in the tree were calculated. Since the breadth of levels may vary considerably across dimensions, the Hilbert order for leaves may not provide good locality for keys higher in the tree which are expressed at higher levels in the dimension hierarchy.

Memory consumption is also a concern. The ingestion throughput gained by using the Hilbert curve comes at the cost of storing a Hilbert index for every node in the tree. With a naïve implementation, this index would take up the same amount of space as the key for a point stored in a leaf node. However, since dimensions can have different breadths, the required amount of space may be significantly lower. Using compact Hilbert indices [37] to...
store only the minimum required number of bits would save space, but could impact locality when MDSs are viewed at different levels.

The Hilbert PDC-tree addresses these issues by first transforming points before performing the Hilbert mapping. Several such transformations have been investigated. The simplest, illustrated in Table 4.3, map the ID directly or simply mask off the most significant bits which represent the dimension.²

<table>
<thead>
<tr>
<th>Dim</th>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 3</th>
<th>Level 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>m</td>
<td>ji</td>
<td>gfe</td>
<td>dcba</td>
</tr>
<tr>
<td>10</td>
<td>nm</td>
<td>i</td>
<td>e</td>
<td>ba</td>
</tr>
<tr>
<td>Direct</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0m</td>
<td>00ji</td>
<td>0gfe</td>
<td>dcba</td>
</tr>
<tr>
<td>10</td>
<td>nm</td>
<td>000i</td>
<td>000e</td>
<td>00ba</td>
</tr>
<tr>
<td>Dimensionless</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0m</td>
<td>00ji</td>
<td>0gfe</td>
<td>dcba</td>
</tr>
<tr>
<td>00</td>
<td>nm</td>
<td>000i</td>
<td>000e</td>
<td>00ba</td>
</tr>
</tbody>
</table>

Table 4.3: Simple ID mappings. Occupied bits which may be set in the ID are labelled by position starting with ‘a’.

These simple mappings have the issue that levels in different dimensions may not have the same breadth, and thus not map to a comparable range in the Hilbert ordering. The more sophisticated mappings shown in Table 4.4 spread or expand values in a given level to match the range of that level in any dimension. The spread mapping simply left-pads levels with zeroes to occupy the number of bits required for that level in any dimension. The expanded mapping uses the same number of bits, but shifts values left and right-pads with zeroes if necessary, ensuring values in any dimension have a similar range.

For example, the ID in Figure 4.3 uses 4 bits for level 4 in dimension 1, but only 2 bits for the same level in dimension 2. The spread mapping compensates by shifting the value for level 4 in dimension 2 left by 2 bits, causing values to span roughly the same numerical range as those in dimension 1. Note that this transformation is only performed on a copy of the key used to calculate the corresponding Hilbert number, the keys stored in the tree for querying are not modified.

These mappings attempt to preserve the hierarchical nature of IDs across dimensions while compensating for dimension skew and reducing space. An alternative approach is to simply eliminate all unused bits and compress IDs as much as possible in each dimension, as

² The Hilbert mapping algorithm implicitly knows the dimensions of its input values, so it is not necessary to include this in the values themselves.
<table>
<thead>
<tr>
<th></th>
<th>Pad</th>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 3</th>
<th>Level 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spread</td>
<td>00000</td>
<td>0m</td>
<td>ji</td>
<td>gfe</td>
<td>dcba</td>
</tr>
<tr>
<td></td>
<td>00000</td>
<td>nm</td>
<td>0i</td>
<td>00e</td>
<td>00ba</td>
</tr>
<tr>
<td>Expanded</td>
<td>00000</td>
<td>m0</td>
<td>ji</td>
<td>gfe</td>
<td>dcba</td>
</tr>
<tr>
<td></td>
<td>00000</td>
<td>nm</td>
<td>i0</td>
<td>e00</td>
<td>ba00</td>
</tr>
</tbody>
</table>

Table 4.4: Hierarchy-preserving compressed ID mappings

shown in Table 4.5. The compressed mapping differs from the others in that the breadth of levels in other dimensions is ignored, so, for example, if the values in dimension 1 require 8 bits, then that dimension will use the 8 least significant bits, regardless of the space required by other dimensions, without any padding inserted. Each dimension thus uses the minimum number of least significant bits necessary.

<table>
<thead>
<tr>
<th></th>
<th>Pad</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compressed</td>
<td>000000</td>
<td>mjigfedcba</td>
</tr>
<tr>
<td></td>
<td>000000</td>
<td>0000nmieba</td>
</tr>
</tbody>
</table>

Table 4.5: Fully compressed ID mapping

The different mapping approaches affect performance as well as the amount of space used to store the resulting Hilbert index. The performance characteristics of the resulting trees are investigated experimentally in Section 4.8, but the amount of space required for given set of dimension hierarchies is straightforward to calculate. For example, the number of bits used for a point by each mapping on a TPC-DS dimension hierarchy at scale 10 is shown in Figure 4.4.

![Figure 4.4: Number of bits used for various Hilbert mappings](image)

4.5.2 Insertion Algorithm

Once the Hilbert index \( h \) for the element to be inserted has been calculated, insertion proceeds similarly to insertion in a linearly ordered tree. However, since parallel execution
of insertions and queries is supported, care must be taken to lock nodes while they are modified, and gracefully handle modifications made by other threads where necessary.

In particular, directory nodes may be split by another thread during an insertion. Algorithm 4.1, select_parent_node, is a utility that handles this situation. Given a Hilbert index \( h \), it selects from an array the node where \( h \) should be inserted. Then, if the node has been split, the algorithm recurses to choose the appropriate side. Though it is highly unlikely for this recursion to continue, this guarantees that the returned node has not been split.

**Algorithm 4.1: select_parent_node**

<table>
<thead>
<tr>
<th>Input: ( P ), an array of directory nodes sorted by Hilbert index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: ( h ), a Hilbert index</td>
</tr>
<tr>
<td>Result: The locked directory node where ( h ) should be inserted</td>
</tr>
</tbody>
</table>

1. Let \( p = \) the first node in \( P \) where \( \text{hindex}(p) \geq h \), locked \( \quad \) (Select parent)
2. if \( p \) has been split
3. Let \( S = \text{descendants}(p) \) \( \quad \) (Get the split parent’s new “sides”)
4. unlock(\( p \)) if it was locked on line 1
5. return select_parent_node(\( S, h \)) \( \quad \) (Choose correct side)
6. return \( p \)

Two variants of this algorithm are required: one for arrays of locked nodes, and one for arrays of unlocked nodes. The latter variant locks nodes individually during the search on line 1. For brevity, one general sketch of this algorithm is shown here and it is assumed that the correct variant is used.

Using this utility, Algorithm 4.2, tree_insert, inserts a new item into the tree. In the simple case where no splitting is required, the algorithm is relatively straightforward. The downward search is performed on a copy of the directory node’s child array after unlocking to minimize contention. Directory nodes are expanded on the way down, and unlocked before proceeding. Thus, the node’s cached aggregate value temporarily does not equal the aggregate value of all its children, but this does not affect correctness since cached aggregate values are only used when a query contains a node’s bounding box completely. Once a leaf level parent is reached, the new item is inserted as a child and the insertion is complete.

If a split is necessary, the parent node is first split to create new “sides”. This is done in a persistent/functional style without modifying the node itself, other than to set its split descendant pointers to the resulting sides. The new item is then inserted as a child of the
Algorithm 4.2: tree_insert

Input: $k$, the point key for the new data item
Input: $v$, the measure values for the new data item
Input: $h$, the Hilbert index of $k$

1. Let $p = \text{root}()$ (Get the locked root node)
2. Expand $p$ to include $k,v,h$
3. while the children of $p$ are not leaves do (Find parent for new leaf node)
   1. Let $C = \text{a copy of children}(p)$
   2. unlock($p$)
   3. Let $c = \text{select_parent_node}(C,h)$ (Select appropriate child)
   4. Expand $c$ to include $k,v,h$
   5. $p := c$ (Move to child and continue search)
4. Let $n = \text{a new leaf node for } k,v,h$
5. if $p$ is not full then
   1. Insert $n$ into children($p$)
   2. unlock($p$)
6. else
   1. Let $S = \text{split_node}(p)$ (The new nodes created by splitting $p$)
   2. Let $s = \text{select_parent_node}(S,h)$ (Select appropriate side)
   3. Insert $n$ into children($s$)
   4. Expand $s$ to include $k,v,h$
   5. split_up($p,S$) (Propagate split upwards if necessary)

appropriate side. Finally, the original parent node is replaced with the new sides in its parent, propagating the split upwards as required. The split_node and split_up algorithms are described in more detail in the following sections.

4.5.3 Selecting a Split Index

The split algorithms of the DC-tree and PDC-tree can not be used in the Hilbert PDC-tree since they do not preserve the order of children nodes. Child order must be preserved in the Hilbert PDC-tree since children are ordered by their maximum Hilbert number.

The Hilbert R-tree also has this restriction, and simply splits nodes in half like a B-tree. This achieves good data balance, but at the cost of increased overlap, since choosing to split in the middle may result in significantly higher overlap than choosing to split at some other index. For a disk-based structure with few dimensions, prioritizing balance
improves space utilization, so for some applications this is a reasonable choice. However, for an in-memory structure with many dimensions, high overlap can dramatically degrade performance, particularly for applications that perform large aggregate queries.

Given an operation to calculate the overlap between two boxes, the overlap that would result from splitting a node at a given index \( i \) is simple to calculate: if \( L_i \) is the union of all boxes to the left of \( i \), and \( R_i \) is the union of all boxes to the right of \( i \), then the overlap resulting from splitting at index \( i \) is simply \( \text{overlap}(L_i, R_i) \).

An analysis of the tree structure resulting from loading various data sets confirms that always splitting nodes in half is often a poor strategy. Though occasionally splitting in half works well, as in Figure 4.5a, many nodes are more problematic, such as the one shown in Figure 4.5b. Here, nearby indices 12 and 20 have less overlap with little cost in balance compared to an even split. Depending on how much balance it is reasonable to sacrifice for overlap, splitting at index 24 or beyond may be a better choice.

![Histograms showing overlap at each split point in observed directory nodes](image)

(a) A node with a good, but sub-optimal, split.  
(b) A node with no clearly ideal split.

Figure 4.5: Overlap at each split point in observed directory nodes

The Hilbert PDC-tree handles these situations by introducing a parameter for the minimum split size. The split algorithm can then examine the overlap at each valid position and choose the one with the least overlap. Ties are broken by choosing the most balanced split, or the smallest key representations, in that order. This behaviour with a fixed fanout of 32 is shown in Figure 4.6. An even split at index 16 is the most common individual choice, but overall, other indices prove better in the majority of cases. However, notice that in Figure 4.6b there is a large peak in overlap produced by even splits. This is due to nodes with a flat “overlap surface” which force the algorithm to resort to the most balanced split.
The use of supernodes which have higher fanout than usual can improve this situation, as in the X-tree. Though the Hilbert PDC-tree uses a completely different split algorithm, the same principle can be applied: if a suitably good split can not be found, then avoid splitting and increase the node size instead.

The X-tree and its direct descendants do not limit the maximum size of supernodes, since a linear scan of even a very large supernode is likely less expensive than descending to several overlapping child nodes. However, in a parallel context, extremely large nodes are problematic since the increased locking and copying overhead limits speedup. Accordingly, the Hilbert PDC-tree adds a maximum fanout parameter which limits the size of a supernode. If a node reaches this size, it is split regardless of the quality of the best possible split.
Figure 4.7 shows the split frequencies and corresponding total overlap for a tree with maximum fanout of 128 (4 times the normal fanout of 32). Comparing with Figure 4.6, the distribution of splits is very similar with a peak at the balanced split point of 16, though a few supernode splits well beyond the normal fanout can be seen. However, the resulting overlap shown in Figure 4.7b is considerably lower than without supernodes as in Figure 4.6b.

The best split position can be found with a linear number of overlap calculations by scanning each potential split index in order. To compute the overlap, the left and right MDSs that would result from a split at that index are required. Let $C$ be the array of $n$ MDSs corresponding to the child nodes. First, an array $R$ of the right side MDSs is computed, where $R[i] = \bigcup_{j=i}^{n-1} C[j]$. This is done in $\Theta(n)$ steps by starting with $R[n-1] = C[n-1]$ and working left, setting $R[i] = C[i] \cup R[i+1]$. The algorithm then proceeds left to right, similarly computing the left-hand MDS $l$ at each step. For each split index $i$, $l$ and $R[i+1]$ can be used to compute the overlap and key sizes resulting from a split at $i$. The balance is trivially computed from $i$ and the tree fanout.

Each position is evaluated according to these criteria, and the best is updated if a better split is found. However, a split index is only considered the “best” if it is a local minimum with respect to overlap, or has zero overlap. This avoids always choosing the first or last valid split index when the potential splits have a hill-like shape as shown in Figure 4.8. In these situations, a supernode is created since it is likely that a better split can be found slightly outside the usual fanout range.

![Figure 4.8: A node with no local minimum split](image-url)
4.5.4 Splitting Directory Nodes

Algorithm 4.3, split_node, outlines the algorithm for splitting a single directory node. Details such as MDS level adjustment and memory management are omitted for brevity.

**Algorithm 4.3: split_node**

**Input:** $n$, a locked directory node to split

**Result:** An array of new locked directory nodes which share the children of $n$

1. Let $C = \text{children}(n)$
2. Let $s =$ the best split index in $C$
3. if $s$ is a bad split and $n$ is not too large then
   4. return $[n]$ \hspace{1cm} (Avoid split, $n$ is a supernode)
5. Let $l, r =$ new directory nodes for the left and right side, locked
6. parent($l$), parent($r$) := parent($n$)
7. descendants($n$) := $[l, r]$
8. foreach locked $c \in C[0]...C[s - 1]$ do
   9. Append $c$ to children($l$) and expand $l$
   10. parent($c$) := $l$
   11. if $c = C[s - 1]$ then
      12. hindex($l$) := hindex($c$)
8. foreach locked $c \in C[s]...C[\text{children($n$)}]$ do
   13. Append $c$ to children($r$) and expand $r$
   14. parent($c$) := $l$
   15. if $c = C[\text{children($n$)}]$ then
      16. hindex($r$) := hindex($c$)
18. return $[l, r]$

When a directory node is split, new sides are created and its children’s parent pointers are updated, but the original node is not modified. This leaves the tree in a safe state for use by other threads: queries which reach the node from above can descend to the children if necessary, and splits which reach the node from below will discover that the node has been split and likely that splitting is no longer necessary. Thus, the split algorithm can safely unlock nodes when moving upwards. Algorithm 4.4, split_up, continues splitting upwards as necessary, taking care to ensure that parent nodes are always locked before their children.

The special case of splitting the root is handled by Algorithm 4.5, split_root. Note that this case uses more coarse grained locking than usual, in order to prevent concurrent
Algorithm 4.4: split_up

Input: \( n \), a locked node that has been split

Input: \( S \), an array of locked nodes to replace \( n \), sorted by Hilbert index

Result: The topmost modified or new directory node

1. Let \( p = \text{parent}(n) \)
2. if \( p = \emptyset \) then \( (n \text{ is the root, grow up}) \)
   - return \( \text{split\_root}(n,S) \)
3. unlock\((n,S)\) \( (Unlock \text{ all nodes}) \)
4. with locked \( p \) do
5.   if \( p \) has been split then
6.     Let \( T = \text{descendants}(p) \)
7.   else if \( \text{children}(p) \) has enough space for \( S \) then
8.     Replace \( n \) in \( \text{children}(p) \) with the nodes in \( S \)
9.     return \( p \)
10. else
11.     Let \( T = \text{split\_node}(p) \) \( (Split \text{ parent into locked sides } T) \)
12.     foreach locked \( s \in S \) do
13.       Let \( t = \text{select\_parent\_node}(T, \text{hindex}(s)) \)
14.       Insert \( s \) into \( \text{children}(t) \) and expand \( t \)
15.       parent\((s) \) := \( t \)
16.     return \( \text{split\_up}(p,T) \) \( (Continue \text{ upwards}) \)

root splits. The root node’s lock is used to protect the tree’s root node pointer, and the new side nodes in \( S \) are not unlocked until the new root has been installed to guarantee that other threads can not attempt to split the root concurrently.
Algorithm 4.5: \texttt{split}\_\texttt{root}

\begin{enumerate}
\item \textbf{Input:} \( n \), the locked root node, which has been split
\item \textbf{Input:} \( S \), an array of locked nodes to replace \( r \), sorted by Hilbert index
\item \textbf{Result:} The new root node
\end{enumerate}

1. Let \( r = \) a new node with children \( S \)
2. \textbf{foreach} \( s \in S \) \textbf{do}
3. \hspace{1em} Expand \( r \) to include \( s \)
4. \hspace{1em} \( \text{parent}(s) := r \)
5. Atomically set \( r \) as the new root of the tree
6. \texttt{unlock}(\( n, S \)) \hspace{1em} \( (Unlock \ all \ nodes) \)
7. \textbf{return} \( r \)

The process of inserting into a Hilbert PDC-tree where splitting is required is illustrated in Figures 4.9...4.12.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure4.9.png}
\caption{Tree insertion step 1: descend and update}
\end{figure}
Figure 4.10: Tree insertion step 2: split parent

Figure 4.11: Tree insertion step 3: split grandparent

Figure 4.12: Tree insertion step 4: split root
4.6 Querying

The query algorithm for the Hilbert PDC-tree, shown in Algorithm 4.6, is similar to that of the PDC-tree. However, the query algorithm does not walk back up the tree to find and resolve concurrent insertions as in the PDC-tree.3

Note that the query algorithm locks only a single node at a time, in order to read the key and children of that node. Like the insertion algorithm, the search proceeds using a copy of the node’s children array after the node has been unlocked to minimize contention.

Algorithm 4.6: tree_query

Input: \( q \), the query MDS
Input: \( n \), a directory node in the tree
Result: The sum of all values within \( q \) in the subtree rooted at \( n \)

1. \( \text{lock} (\text{node}) \)
2. Let \( r = 0 \) \hspace{1cm} (Result)
3. switch \( \text{mds} \_\text{coverage} (q, \text{key} (\text{node})) \) do
   4. case FULL do
      5. \( r := r + \text{data} (n) \)
      6. \( \text{unlock} (n) \)
   7. case PARTIAL do
      8. Let \( C = \) a copy of \( \text{children} (n) \)
      9. \( \text{unlock} (n) \)
      10. if \( C \) points to directory nodes then
          11. foreach \( c \in C \) do
              12. \( r := r + \text{tree} \_\text{query} (q, c) \)
          else
              13. foreach \( c \in C \) do
                  14. if \( \text{mds} \_\text{contains} (q, \text{key} (c)) \) then
                      15. \( r := r + \text{data} (c) \)
          end
      case NONE do
          17. \( \text{unlock} (n) \)
   19. return \( r \)

3. Section 4.7 describes the correctness model of the Hilbert PDC-tree in more detail.
4.7 Correctness

Deadlock freedom is achieved by guaranteeing that concurrent locks are always taken in the same order. The approach used here is similar to the concurrent B-tree of Lehman and Yao [56], though the ordering and split strategies differ. The lock ordering used in the Hilbert PDC-tree is given by Definition 4.3.

**Definition 4.2 Node version**

The *version* of a tree node \( a \), denoted \( \text{version}(a) \), is the number of split ancestors of \( a \). For example, the initial root of an empty tree has version 0, if it is split, then the resulting sides have version 1, and so on.

**Definition 4.3 Node lock ordering**

For two arbitrary tree nodes \( a \) and \( b \), the lock ordering \( a < b \) is:

- *Top down*: If \( a \) and \( b \) are at different levels of the tree, then \( a < b \) if and only if \( a \) is at a higher level of the tree than \( b \).
- *Oldest first*: If \( a \) and \( b \) are on the same level of the tree and have different ages, then \( a < b \) if and only if \( \text{version}(a) < \text{version}(b) \).
- *Left to right*: If \( a \) and \( b \) are on the same level of the tree and have the same age, then \( a < b \) if and only if \( a \) is to the left of \( b \).

It must be shown that all concurrently held locks are acquired in this order. First, this property is shown for the individual utility algorithms.

**Lemma 4.1 Node selection lock order**

For two tree nodes \( a \) and \( b \), Algorithm 4.1, `select_parent_node`, locks \( a \) followed by \( b \) if and only if \( a < b \).

*Proof.* There are two cases to consider where `select_parent_node` is used:

1. When called given the split descendants of a node, the initial search may lock an immediate split descendant of that node, which is in *oldest first* order.
2. When called given the children of a node, the initial search may lock a child of that node, which is in top down order.

In either case, if the initial search finds a split node, then the node is unlocked and split descendants of that node are locked as necessary, which is in oldest first order.

\[\text{Lemma 4.2 Split node lock order}\]

For two tree nodes \(a\) and \(b\), Algorithm 4.3, `split_node`, locks \(a\) followed by \(b\) if and only if \(a < b\).

\[\text{Proof.}\] The input node \(n\) is locked by the caller. If no good split is found, then the algorithm immediately returns and no locks are acquired. Otherwise, the new sides \(l\) and \(r\) are created as split descendants of \(n\), and locked in left to right order on creation, in oldest first order with respect to \(n\). Finally, the children of \(n\) are individually locked to distribute between \(l\) and \(r\), which is in top down order since \(n, l,\) and \(r\) were locked earlier.

\[\text{Lemma 4.3 Split up lock order}\]

For two tree nodes \(a\) and \(b\), Algorithm 4.4, `split_up`, locks \(a\) followed by \(b\) if and only if \(a < b\).

\[\text{Proof.}\] If the split node \(n\) is the root, then this special case is handled by `split_root`, for which the lemma trivially holds since no further locks are acquired.

Otherwise, all nodes are first unlocked, then the parent node \(p\) is locked. In the case where \(p\) is full and not yet split, `split_node(p)` is used, which locks the nodes in \(T\) in the correct order (Lemma 4.2).

Then, each side \(s\) is locked in left to right order, which is in top down order with respect to \(T\). After distributing the children, the algorithm recurses, and either splits the root or continues splitting upwards. In either case, all locks are released before further locks are acquired.

Given the above, it can be shown that the tree insertion algorithm itself follows the correct lock order:
Lemma 4.4 Tree insert lock order
For two tree nodes $a$ and $b$, Algorithm 4.2, \texttt{tree\_insert}, locks $a$ followed by $b$ if and only if $a < b$.

Proof. The initial search begins by locking the root, and proceeding downwards, unlocking the current node before descending. This locks a single node at a time in \textit{top down} order until the parent $p$ is found. If a split node is encountered, then \texttt{select\_parent\_node} will find the latest appropriate split descendant, locking nodes in \textit{oldest first} order (Lemma 4.1). Then, if $p$ is not full, the algorithm terminates without taking any further locks.

Otherwise, $p$ is split, which acquires locks in \textit{oldest first} and \textit{left to right} order (Lemma 4.2).

Then, a parent from $S$ is selected with \texttt{select\_parent\_node}, which does not acquire any locks since the nodes in $S$ are already locked and do not have any split descendants. After a new item is inserted into the appropriate side $s$, the split is propagated upwards with \texttt{split\_up} which only locks in a correct order (Lemma 4.2). The algorithm then terminates.

The case for querying is trivial:

Lemma 4.5 Tree query lock order
For two tree nodes $a$ and $b$, Algorithm 4.6, \texttt{tree\_query}, locks $a$ followed by $b$ if and only if $a < b$.

Proof. The lemma trivially holds, since the algorithm takes at most one lock at a time, in \textit{top down} order.

Finally, the result for the entire tree can be stated:

Theorem 4.6 Tree deadlock freedom
The Hilbert PDC-tree is free of deadlocks.

Proof. The interface of the tree consists of Algorithm 4.2 and Algorithm 4.6, both of which take concurrent locks according to a single well-defined order (Definition 4.3), as shown in Lemma 4.4 and Lemma 4.5, respectively. Therefore, the Hilbert PDC-tree is free of deadlocks and may safely be used by several threads.
The consistency guarantee provided by the Hilbert PDC-tree is given by Theorem 4.8. Note that this is a weaker guarantee than the PDC-tree, since VOLAP does not require the model provided by the PDC-tree and the relaxation allows for higher throughput due to reduced thread contention and a simpler query algorithm.

First, Lemma 4.7 states that a value inserted into the tree is never lost by subsequent insertions.

**Lemma 4.7 Tree insert consistency**

For an arbitrary node \( n \), once an insert operation \( I \) descends below \( n \), the value of \( I \) will be included in \( n \) or some split descendant of \( n \).

**Proof.** Note that the aggregate value of \( n \) is updated during the initial downward search. In the case where no splits occur, then, the theorem trivially holds.

If \( n \) is subsequently split, then \( I \) will be included in one of the resulting sides \( l \) or \( r \), and so on if \( l \) or \( r \) are further split. Since a node is always locked when split, and the `split_up` algorithm always finds the most recent suitable split descendant while ascending the tree, replacing a node in its parent’s child array with its split descendants can never result in \( I \) being lost.

In the case where the root is split, the new root node is updated to include all values of the old root node before the root-level locks are released, serializing root splits and ensuring that \( I \) is not lost.

Therefore, in all cases, once \( I \) descends below \( n \), its value will always be included in some split descendant of \( n \). \( \square \)

Given this, the consistency guarantee of the Hilbert PDC-tree, Theorem 4.8, is relatively simple to prove:

**Theorem 4.8 Tree consistency**

Let the *start* of any tree operation be the time it acquires the initial lock on the root node, and the *end* be the time it releases its final lock. Then, any query \( Q \) that starts after an insert \( I \) ends includes the value of \( I \) if necessary.

**Proof.** The lock taken on the root node is the serialization point that defines operation order.
After $I$ ends, its value is included in the root of the tree (Lemma 4.8). Therefore, the result of $Q$ will include the value of $I$ if it is within the query region.

Note that though this is a relatively weak guarantee, in practice an insert will be included in queries that start well before the insert finishes, and often in those that start shortly after the insert starts. However, this guarantee is all that is required for VOLAP to provide serializable execution, as explained in Section 5.3.5.

### 4.8 Experimental Evaluation

To evaluate the performance of the Hilbert PDC-tree, experiments were performed using the TPC-DS [63] data set with scale factor 50 on an Intel Core i7-3770 (4 cores, 8 threads) with 32 GiB of RAM. These experiments use 8 hierarchical dimensions: *item*, *store*, *sold_date*, *sold_time*, *addr*, *cdemo*, *hdemo*, and *customer*.

Six tree variants are tested: the five Hilbert mappings described in Section 4.5.1, and a PDC-tree implementation based on the same underlying tree representation and implementation to ensure a fair comparison. Each is evaluated on streams consisting of only inserts, only aggregate queries, and a 50% mix of inserts and aggregate queries. The latencies shown are the average for many operations, where each individual test at a given size is run until at least 20 queries are executed and at least 4 seconds have passed.

Test queries are randomly generated before benchmarking as described in Appendix B.2 and placed into bins according to their coverage. The generated queries include several possibly non-contiguous values at different levels in all dimensions. Queries range from very low coverage to aggregations of nearly the entire data set.

Fanout is configured to 32, with a maximum fanout of 128 for supernodes. Except where explicitly shown, the values shown are for 8 threads, which showed the best overall performance on the test platform.

#### 4.8.1 Insertion

Performance for a stream consisting entirely of insertions is shown in Figure 4.13. It is clear that the Hilbert PDC-tree achieves a much higher rate of ingestion than the PDC-tree. The various Hilbert mappings (explained in Section 4.5.1) vary in throughput by about 100 thousand inserts per second, but even the slowest (expanded) at approximately 250 thousand
inserts per second is over 12 times faster than the PDC-tree at approximately 20 thousand
inserts per second. The fastest Hilbert mapping for ingestion (direct) sustains approximately
300 thousand inserts per second, around 15 times faster than the PDC-tree. This performance
edge is maintained as data size increases, with all trees showing a relatively flat trend.

The performance discrepancy between the various Hilbert mappings is partially due
to the expense of manipulating the keys before applying the Hilbert mapping. The direct
mapping is the fastest, since no copying or manipulation is required at all. The expanded and
compressed mappings are slower since more bit shifting is required, as well as dictionary
searches to find the appropriate range of bits for a level. Further optimization of the Hilbert
mapping could reduce this gap and improve overall ingestion performance, but the cost
of Hilbert mapping is very low compared to the geometric calculations performed by the
PDC-tree. This is the primary reason why the Hilbert PDC-tree can sustain a much higher
rate of ingestion.

![Graph](image)

Figure 4.13: Performance for a stream of inserts

### 4.8.2 Querying

Performance for a stream consisting entirely of aggregate queries is shown in Figure 4.14.
The difference in performance between the various Hilbert mappings is apparent, with the
expanded mapping performing better than all other configurations. The expanded mapping
performs best because it spreads each level of each dimension evenly throughout the Hilbert
space, regardless of the range actually covered by the data. This results in a tree structure
that is more resilient to the order in which elements are inserted. However, this mapping
is also the slowest during ingestion, so there is a slight trade-off between insert and query performance where the best choice may differ depending on the application.

It is particularly apparent that the compressed mapping, which uses a minimal amount of space for Hilbert indices, performs poorly. This is because the compressed mapping does not preserve the hierarchical nature of IDS at all, so locality is not well-preserved with higher level keys which is particularly significant for large aggregations. Compared to the PDC-tree, all other Hilbert PDC-tree variants perform significantly better, though the improvement is not as dramatic compared to insertion.

Figure 4.14: Performance for a stream of queries

Performance for a mixed stream of 50% inserts and 50% queries is shown in Figure 4.15. These timings are very similar to the query-only timings shown in Figure 4.14, since the overall time is dominated by queries. Performance is slightly better than the query-only workload, which confirms that the minimal locking scheme keeps thread contention low and prevents concurrent inserts from significantly harming query performance.

A breakdown of the best Hilbert PDC-tree variant (expanded) and the PDC-tree performance with respect to query coverage is shown in Figure 4.16. These figures are for a stream of only queries, chosen from 3 bins (one for each coverage range) uniformly at random. For this data set, the Hilbert PDC-tree outperforms the PDC-tree for all query coverages. The best improvement is seen for queries with medium to high coverage, since a good tree structure allows high coverage queries to better utilize cached aggregate values and avoid traversing many large subtrees.

Note that these results show the mean performance for various queries that have very
different performance characteristics. Consequently, error bars have been omitted to avoid obscuring the performance trends. However, analysis of the data confirms that the worst-case query performance of the Hilbert PDC-tree is better than that of its predecessor. For example, with 8 threads, the slowest of all queries for the PDC-tree took approximately 10.5 seconds, while that of the Hilbert PDC-tree using the expanded mapping took only 5.7 seconds. With the exception of the compressed mapping, the worst query time for the Hilbert PDC-tree was faster than that of the PDC-tree by a significant margin in all cases.

4.8.3 Speedup and Hilbert Mappings

The throughput speedup for a mixed stream of 50% inserts and 50% aggregate queries is shown in Figure 4.17.

The best speedup occurs when using 8 threads, which corresponds to the number of
hardware-supported threads on this processor. Speedup for the best mapping peaks at just under 4, the number of physical cores present. This is to be expected, since this test is memory-bound enough to benefit from hardware-supported threads, but compute-intensive enough to be bound by the amount of actual parallel execution available.

Beyond 8 threads, speedup flattens, since no more parallelism (real or virtual) is available in hardware, and there is no access to external resources (such as disks or the network) which might benefit from further overloading the scheduler. Increase in speedup roughly corresponds to the query performance seen by each tree variant, since query time dominates as shown in the previous section.

Note that the root lock of the tree is a common point of contention, but as threads descend the tree along different paths, they no longer contend for the same locks. Thus, depending on the data set and query workload, similar results can be expected for higher core counts until the number of cores approaches the fanout of the tree, at which point reduced speedup would be expected due to more contention.

![Figure 4.17: Speedup for a mixed stream of inserts and queries](image)

The benefits of the various Hilbert mappings depend on several factors, including data distribution. However, generally, the expanded mapping is the best choice for query performance, while the spread mapping is a better choice if ingestion throughput is the top priority since it supports a higher rate of ingestion with only a modest penalty in query performance.
4.8.4 Dimensionality

To evaluate how the Hilbert PDC-tree scales as the number of dimensions increases, experiments were run using Zipf-distributed data with a moderate skew of 0.5, as described in Appendix B. For comparison, two R-tree variants were included: one using the classic insertion algorithm (R-tree), and the other using Hilbert ordering (Hilbert R-tree). All trees, including the R-tree variants, are based on the same underlying tree implementation, so all benefit from parallelism and the same optimizations.

The various trees are compared by executing the same queries. Since the generated test queries are MDSs that may include non-contiguous values, it is not possible to directly convert queries to a single MBR for testing R-tree variants. Instead, each query MDS is converted into the equivalent set of MBRs, and the total time to execute all the resulting MBR queries is measured. This compares the trees fairly by performing an equivalent task, that is, the results obtained from each tree are identical.

With few dimensions, the R-tree variants perform better since MBRs are less expensive to work with. However, as the number of dimensions increases, the benefits of the MDS-using PDC-trees become clear. Past 8 dimensions, the query performance of the R-tree variants degrades dramatically. Queries beyond 20 dimensions take an unreasonably long time to complete for the R-tree variants, so these experiments were stopped at this point.

The Hilbert PDC-tree preserves the PDC-tree’s ability to gracefully scale to a large number of dimensions, though the query performance advantage seen in the (8 dimension) TPC-DS results narrows as the number of dimensions increases. However, the improvement in insert performance is not only preserved, but becomes greater with more dimensions. This is because the PDC-tree must perform more expensive geometric calculations as the number of dimensions increases, but the insertion algorithm for the Hilbert PDC-tree is based on a linear ordering. As the number of dimensions increases, the cost of mapping keys to Hilbert indices and the cost of comparing Hilbert indices increases slightly, but this overhead is very small compared to the additional work the PDC-tree must perform. Consequently, the Hilbert PDC-tree shows almost no increase in insert latency as the number of dimensions increases, but inserting into the PDC-tree becomes significantly more expensive.
4.9 Summary

The Hilbert PDC-tree is a parallel index structure for data with many hierarchical dimensions that supports a much higher rate of data ingestion than comparable structures. This is primarily achieved by avoiding geometric calculations during insertion by instead inserting records based on the Hilbert index of their keys.

Though earlier structures have applied a similar idea, the use of hierarchical data with many dimensions introduces unique difficulties. In particular, earlier Hilbert-ordered structures can not effectively manage hierarchical data, and earlier structures designed for hierarchical data use split algorithms that can not be used in conjunction with Hilbert ordering. The Hilbert PDC-tree introduces a hierarchy-aware Hilbert transformation, and a new overlap-minimizing split algorithm that maintains an efficient tree structure for aggregate queries within the constraints imposed by the Hilbert ordering.

Experiments confirm that the Hilbert PDC-tree ingests data at a much higher rate than its closest ancestor, the PDC-tree, while providing good performance for queries that aggregate large fractions of the data stored in the tree. Despite the large increase in ingestion throughput, experiments confirm that the structure efficiently supports very large aggregate queries with comparable performance to trees constructed with the much more expensive PDC-tree insertion algorithm. Finally, the Hilbert PDC-tree scales well to many more dimensions than R-tree variants can effectively support.
Chapter 5
VOLAP: A Distributed System for Real-Time OLAP

5.1 Introduction

The Hilbert PDC-tree provides fast data ingestion and aggregate querying in a multi-core environment, but the amount of data that can be realistically stored on a single multi-core machine is inherently limited. This chapter describes VOLAP, a fully distributed system that uses the Hilbert PDC-tree as a primitive to support much larger databases and many more simultaneous users than a single multi-core machine could handle.

VOLAP partitions data into shards stored in trees on worker nodes. Multiple server nodes handle incoming streams of insertions and aggregate queries, and route them to the appropriate workers. A manager background process monitors the state of the system, stored in Zookeeper [45], and initiates real-time load balancing operations as necessary. This load balancing is fully automatic and dynamically adapts to the data distribution, which can change significantly over time due to the high-velocity of incoming new data.

Clients interact with VOLAP via server nodes, which handle incoming streams of insertions and aggregate queries, and route them to the appropriate workers. Each client is attached to a single server node, though many server nodes may exist to serve a large number of clients, as illustrated in Figure 5.1.

Serializable [12] execution, that is, execution of an operation stream which is functionally identical to a serial execution, is particularly challenging in a highly parallel system like VOLAP. In order to accommodate several high-velocity clients, VOLAP provides different guarantees in different contexts. VOLAP can guarantee that the execution of an individual client stream is serializable, so that any query includes all relevant results from any insertion issued earlier in the stream, and none issued later. However, if a high-throughput application does not require serializable execution, this can be disabled to improve performance.

\footnote{This chapter is a more detailed description of previously published work [26].}
Between client sessions, VOLAP provides best-effort freshness and aims to minimize the time required for an insertion to be included in later queries issued by other clients. Clients attached to the same server observe a smaller delay than those attached to different servers since inter-server synchronization is not required for one client’s requests to be affected by the other.

VOLAP is fully scalable and designed for an elastic cloud computing environment. Both server and worker nodes can be added or removed as necessary to adapt to the current workload. Unlike many other systems, no single node acts as a performance bottleneck or point of failure for the entire system, and there is no special partitioning dimension that needs to be manually configured.

5.2 Related Work

Many published systems store and query large data sets in distributed environments. Hadoop [3] and its file system HDFS are popular examples, with applications typically built on the MapReduce [22] model. However, these systems are not designed for real-time operation. Instead, they are based on batch processing or “quasi real-time” operations [13, 47, 67, 68]. The situation is similar for Hive [69], HadoopDB [1], BigTable [18], BigQuery [34], and Dremel [60].

Storm [70] introduced a distributed computing model that processes streams such as in-flight Twitter data. However, Storm assumes small data packets that can quickly migrate between different computing resources, and is more oriented towards stream processing rather than data warehousing and analytics.

There are various publications on distributed B-trees for cloud platforms [73], however, such 1-dimensional indices are unsuitable for OLAP queries. There have been efforts to build distributed multi-dimensional indices on cloud platforms based on R-trees or related multi-dimensional tree structures [72, 75, 52]. However, these methods do not support dimension hierarchies, which are essential for OLAP queries.

Several more recent cloud-based OLAP systems [17, 20, 38, 58] have been designed which are also based on MapReduce. Though these systems support OLAP, they share the limitations of the above-mentioned systems based on MapReduce and do not support real-time operation.
The systems most related to this work are Druid [74], Brown Dwarf [28], SAP HANA [30], IBM Netezza data warehouse appliance [46], HyPer [49], and CR-OLAP [24]. The remainder of this section discusses these in more detail.

Druid [74] is an open-source, distributed, scalable OLAP store designed for real-time exploratory queries on large quantities of transactional events. Druid shares many goals with this work, in particular being distributed and real-time while serving as a data store and supporting OLAP. However, Druid is specialized to operate on data items that have timestamps, such as network event logs. In particular, it partitions data based on these timestamps and queries are expected to apply to a particular range of time. This limits the system’s usefulness to data sets and workloads that fit this model.

Brown Dwarf [28] is a real-time, distributed, fault tolerant OLAP store which is intended to be used on commodity hardware and uses a decentralized graph to quickly retrieve pre-computed aggregations. However, Brown Dwarf does not fully support dimension hierarchies, and individual queries must specify either a single point in a dimension, or all points in a dimension, which is too restrictive for general purpose OLAP.

SAP HANA [30] is a real-time in-memory database system that also supports OLAP queries. HANA relies mainly on vertical scaling. A basic HANA installation uses a single, special-purpose, very large multi-core compute node. A limited scale-out version for multiple compute nodes is available, using a distributed file system that provides a single shared data view to all compute nodes. Horizontal scalability is restricted, however, because the system has a single master node for maintaining the shared data view, and this single master node becomes a bottleneck as system size increases.

The IBM Netezza data warehouse appliance [46] relies on special-purpose FPGA boards that provide a hardware implementation of OLAP functionality. While special-purpose hardware can provide impressive performance, it does so at extremely high cost, and such systems are more difficult to deploy or scale than those which can make use of commodity hardware or virtual machines.

HyPer [49] is an in-memory database system that supports fast transactions alongside a facility for creating lightweight snapshots for OLAP sessions. HyPer makes use of the operating system’s virtual memory facilities to quickly create snapshots for analysis without copying data unnecessarily. Conceptually, HyPer provides a lightweight on-demand data warehouse, which supports read-only OLAP access to a consistent snapshot of the database.
at a particular point in time. This is ideal for some applications, but less well-suited to those that process a high-velocity stream of mixed insertions and aggregate queries. HyPer is a single-server system, though the snapshot technique it uses may be applicable to distributed systems.

The predecessor of this work, **CR-OLAP** [24], is similar to HANA in that it is also a centralized system with a single master node which can become a bottleneck in larger systems. CR-OLAP uses the PDC-tree [27] as a building block to conceptually form a larger distributed tree, where the top few levels are stored on the master node and subtrees are stored on worker nodes. This centralized design does not allow for a reliable distributed index, which restricts horizontal scalability.

### 5.3 Architecture

The VOLAP architecture is composed of:

- **m servers** $S_1, \ldots, S_m$ for handling client requests (inserts and queries).
- **p workers** $W_1, \ldots, W_p$ for storing data and performing operations requested by servers.
- A **Zookeeper** [45] cluster for maintaining global system state.
- A **manager** background process for analyzing global state and initiating load balancing operations.

Workers and servers are multi-core machines which execute several threads in parallel. All data is kept in main memory, as is typical for current high performance databases. VOLAP is *elastic* in that more workers and servers can be added if necessary. With increasing database size and/or changing network topology, the data is reorganized to make the best use of currently available resources. Figure 5.1 illustrates the distributed architecture, and a summary of the system parameters is given in Table 5.1.

Workers store data and process operations that apply to their locally stored data. The global data set is partitioned into data **shards** $D_1, \ldots, D_n$. Each shard $D_i$ has a **bounding box** $B_i$ which is a spatial region containing $D_i$, represented by either an MBR or MDS. Each worker typically stores multiple shards, and the bounding boxes for different shards may overlap. Bounding boxes may overlap, though an individual data item is stored in only one shard.
Servers receive a stream of OLAP operations from clients, determine the shards relevant to each operation, and forward the operations to the worker(s) responsible for those shards. Once the workers respond, the server reports the result to the originating client.

The manager requires few resources and can be run on any machine, for example on a dedicated coordination node that also serves as a Zookeeper master, or on the same node as a server.

All nodes communicate using ZeroMQ [43], a high-performance asynchronous messaging library designed for scalable distributed applications.

### 5.3.1 System Image

The system image represents the global system state, and is stored in Zookeeper [45], a fault tolerant distributed coordination service. The image contains the global information required by servers and the manager, including:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_i$</td>
<td>Total number of data items</td>
</tr>
<tr>
<td>$N_r$</td>
<td>Number of replicas of each data shard $D_i$</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of data shards $D_i$</td>
</tr>
<tr>
<td>$p$</td>
<td>Number of workers $W_j$</td>
</tr>
<tr>
<td>$m$</td>
<td>Number of servers $S_k$</td>
</tr>
</tbody>
</table>

Table 5.1: System parameters
• Lists of the current workers and servers.
• Configuration parameters.
• For each shard $D_i$:
  – Its size $|D_i|$, that is, the number of data items in $D_i$.
  – Its bounding box $B_i$ which contains all points in $D_i$.
  – The address of the worker(s) where it is located.

Each server maintains a local image which serves as an in-memory cache to prevent Zookeeper from becoming a bottleneck. Given an insertion or query, the server uses the local image to find the relevant shards as well as the address of their corresponding worker(s).

The server updates the global image in Zookeeper at a configurable rate if the local image has changed due to insertion, for example every 3 seconds as in the experiments below. Servers make use of Zookeeper’s watch facility to be notified of changes, and update their local image as necessary. Workers update shard statistics in Zookeeper periodically as well, to allow the manager to plan load balancing operations.

5.3.2 Index Data Structure

Since the local image is responsible for finding the shards relevant to each insertion or query, a fast index structure is crucial for high performance. Two key aspects of the index affect performance: search speed, and the global structure that results from choosing a given shard for an insertion. In particular, overlapping shards increase the likelihood that queries must be sent to many workers.

A modified PDC-tree [27] is used to serve this purpose. The basic structure of the tree is conventional: nodes have a bounding box which encompasses those of all its children. The index tree has exactly $n$ leaves which correspond to the data shards in the system. Each leaf has a shard ID $D_i$ and the bounding box $B_i$ of the corresponding shard.

The server tree exploits the cached aggregate values in the index by storing a set of shard IDs in each node, and using set union as the aggregation function. Directory nodes higher in the tree thus contain the set of all shard IDs in the subtree rooted at that node, as illustrated in Figure 5.2. For example, if the system receives a query for a large box that contains the entire database, only the root node is accessed and no tree traversal is necessary, since the root node contains the set of all shard IDs in the system.
Insertions differ from those in a conventional tree, since the leaves are fixed. The Hilbert ordering is not used here since the linear ordering imposed on leaves is restrictive, particularly when considering load-balancing which requires the ability to explicitly split and migrate shards (as explained in Section 5.5). Additionally, the Hilbert ordering is relatively sensitive to the order in which the first few data items are inserted. The relatively inexpensive split algorithm mitigates this problem for shards, but in this distributed context, leaves may exist on separate machines so imbalance is extremely costly.

Accordingly, an insertion algorithm like that of the PDC-tree is used instead, which chooses a subtree based on geometric comparisons. Though this approach has higher overhead, as explained in Chapter 4, it is much less significant in this case compared to shards, since the index tree is relatively small. The process of walking to a leaf is identical to a normal tree, but when a leaf is reached, its bounding box is expanded and no children are added. Consequently, an insertion never results in a node in the index being split.

Though insertions do not modify the structure of the tree, synchronization with the global image in Zookeeper may require structural modification. Adding a new shard to the system inserts a new leaf, expanding and possibly splitting internal nodes in the process. When a bounding box in Zookeeper expands, the corresponding leaf’s bounding box is expanded accordingly, as well as those of all nodes on the path from the leaf to the root as necessary. If a shard is split, the corresponding leaf is removed and replaced with two leaves which correspond to each side of the split. When a bounding box in Zookeeper expands, the corresponding leaf must be expanded. This operation is unique, since the expansion is a bottom-up operation. Searching for a leaf via its bounding box is expensive since there may be overlap higher in the tree, forcing the search to visit several subtrees of a single
node. Instead, a separate index of pointers to leaf nodes is maintained, keyed by shard ID. When a bounding box expands, the leaf is found immediately using this index, the leaf’s bounding box is expanded, and the expansion is propagated up the tree towards the root as necessary. This operation temporarily violates the tree invariant that a node’s bounding box encompasses all of its children. However, this is not problematic since it is only used for synchronization, and will never cause queries to miss data they would have otherwise included prior to the start of the synchronization.

The index data structure is based on the same underlying tree implementation as the Hilbert PDC-tree, so many operations can be performed in parallel. The server uses many threads, all using the same index in parallel, to ensure high throughput.

5.3.3 Fault Tolerance

VOLAP uses redundant replication of shards to ensure that the system is able to operate without interruption or loss of data in the event of worker node failures. When replication is enabled, each shard is stored on $N_r$ different workers. If a worker fails or stops responding, $N_r - 1$ workers remain available to handle queries on that shard. Shard replicas are kept up to date by sending each insertion from the server to all $N_r$ workers replicating the relevant shard. Since insertions will invariably arrive at different workers at different times, shard replicas will rarely be identical at any one point in time. To mitigate this, quorum consensus is used for insertions and queries. As illustrated in Figure 5.3, the write quorum parameter $W$ dictates the number of worker replies the server must receive before notifying the client that the insert is considered complete. This is similar to how write quorums work in many fault tolerant key-value stores [53, 23]. Similarly, queries are sent from the server to all workers storing replicas of each relevant shard, and $R$ read quorum responses must be received from each relevant shard before the server performs the intermediary aggregation.

Once a server receives $R$ query responses from a set of shard replicas, it must decide which shard aggregation is the best or “most correct” representation of the true result. Since each result is the aggregation of any number of points, timestamps of individual insertions or points can not be used to solve this problem, unlike in key-value stores. VOLAP uses different methods based on the aggregation function of the query to determine which shard replica’s result is closest to the true result. For example, assuming points are never deleted from the system as is typical in OLAP, for monotonically increasing aggregation
functions like \texttt{max} or \texttt{count}, the best query replica result is always the greatest. Likewise, for monotonically decreasing functions like \texttt{min}, the best result is the smallest. For non-monotonic functions like \texttt{sum} and \texttt{mean}, the result with the largest \texttt{count} value is likely the most accurate.

When worker nodes die or become unresponsive, the manager removes the worker from the system image and alerts servers not to send messages to the unresponsive worker. Periodically, the manager checks the system image to ensure that each shard is replicated on $N_r$ workers. If a shard has fewer than $N_r$ replicas, the manager coordinates a copy to a new worker in order to create another replica of the shard. If a “zombie” worker becomes available again, its memory is wiped and it is treated as a new worker which has been added to the system.

Server threads always monitor a local command socket which broadcasts system changes like worker death. If a worker dies while a server is waiting for a reply from that worker, the server adjusts accordingly to ensure execution proceeds. If the worker is a redundant replica, the server removes the worker from its pool of expected responses, and completes the operation and responds to the client if all other workers have responded. If the operation can no longer be completed successfully because too many workers have died, the server aborts the operation and responds with an error.

Higher values of $N_r$ increase the number of workers the system can lose while still maintaining correct operation. Increasing $N_r$ also consumes more resources, as each insertion and query must be executed on $N_r$ different workers. The quorum parameters $W$ and $R$ thus give the user a level of control over the consistency of their queries. Lower values of $W$ and $R$ decrease latency at the cost of decreased consistency, while high values of $W$ and $R$ tune
the system for high data consistency at the cost of insert and query latency.

5.3.4 Multi-Threaded Message Handling

Careful handling of messages is necessary to maintain high performance while ensuring correctness in a highly parallel system like VOLAP. Servers and workers have a similar design for handling incoming requests: a single receiver thread reads messages from the network socket, and several processing threads handle requests. The receiver thread distributes requests to processing threads via local sockets. All sockets are ZeroMQ sockets, using the appropriate transport backend (tcp or inproc). Using sockets for inter-thread communication avoids many synchronization issues and allows for a high degree of parallelism.

The receiver thread balances requests among processing threads by maintaining a list of currently idle threads. When a request is received, an idle thread is popped off the list, and the request is forwarded to the corresponding local socket. When the thread has finished processing the request, it sends the reply back to the receiver (and ultimately the client) via the same socket, followed by a special message to indicate that the thread is once again idle. The processing thread is then placed back on the list of free threads.

In the server, requests typically require sending further requests to workers. One of two possible configurations can be chosen by the user:

- **Thread sockets**: Each processing thread has its own network socket for sending messages directly to workers.
- **Single socket**: Each processing thread has a local socket that forwards to a sender thread, which uses a single network socket to send all messages to workers.

Thread sockets eliminate the hop through the sender thread, and thus can achieve slightly higher throughput. However, the lack of a single ordered stream of messages to workers precludes serializable execution, which requires a single network socket as explained in the following section.

The path of a request through the system using thread sockets can be seen in Figure 5.1. Sm’s receiver receives a request from Cc, and forwards it to thread T1. After local processing, T1 sends a derived request to worker Wp. Similarly, Wp’s receiver receives the request and forwards it to thread T2. T2 accesses shard Dn to handle the request, then responses return along the same path to the originating client Cc.
5.3.5 Serializability

As explained in Section 3.1.2, in the context of VOLAP, a serializable execution requires that any queries include all relevant results from insertions issued prior in the stream, and none issued later. Note that this does not impose an execution order for a series of successive insertions or queries: insertions only affect the outcome of queries, so successive insertions can safely be executed in parallel until a query arrives. Similarly, queries do not affect the database state, so successive queries can be executed in parallel until an insertion arrives. Synchronization is therefore required at any transition from query to insertion or vice-versa, as illustrated in Figure 5.4.

Figure 5.4: The transitions $t_i$ within a stream where synchronization is required

In order to support serializable execution, the single socket configuration described in the previous section is required. The sender thread and single network socket provide the serialization point necessary to control the order in which messages are sent to workers. The architecture of the system with this configuration is shown in Figure 5.5.

Figure 5.5: System overview with send thread for serializable execution

A naïve implementation could simply block processing of the stream entirely at each transition, but this would severely restrict throughput for streams with interspersed insertions
and queries. Instead, VOLAP uses a parallel pipeline model where synchronization occurs at various stages throughout the processing of a request, allowing insertions and queries to be executed in parallel at different stages in the pipeline.

There are several stages in the execution of a request where the correct order must be ensured. At each such stage, if a transition in the stream is encountered, processing must be blocked until the previous operation has completed execution of that stage. This is achieved with a series of iq-barriers (insert-query barriers), which allow any number of insertions, or any number of queries, to execute in parallel, but block processing if an insertion is encountered while queries are being executed or vice-versa.

![Figure 5.6: Path of a request through the system with serialization points](image)

The serialization points encountered while processing a request, as illustrated in Figure 5.6, are:

1. Server Receive: When the request arrives at the server’s receive socket, the first iq-barrier is reached by the (single) receiver thread. Then, the request is forwarded to a server thread for processing via an internal socket.

2. Server Prepare: When a server thread receives the request, it deserializes the request and applies it to the index.

3. Index: After the request is applied to the index, the corresponding worker request(s) is/are enqueued for sending to the relevant worker(s).

4. Enqueue: After all messages have been enqueued to be sent to workers, a special message is also enqueued to signal the sender thread to reach the next iq-barrier.

5. Send: When the sender thread receives the special synchronization message, it knows all worker messages for this request have been enqueued, so serializable delivery of requests to workers has been ensured.
6. Worker Receive: Identical to step 1 in the server.

7. Worker Prepare: Identical to step 2 in the server.

8. Commit: After the request has been applied to all relevant shards, serializable execution has been completed.

In order to maximize throughput, more iq-barriers are used than are strictly necessary to ensure correctness. For example, barrier 2 could be omitted, but its inclusion allows following events to be distributed to server threads and prepared in parallel while the current operation is being committed to the index. This design allows many requests to execute in parallel at different stages, so serializable execution has only a moderate impact on performance, as demonstrated in Section 5.8.2.

These iq-barriers must allow two separate types of non-exclusive lock (read and write), which is not provided by any of the standard synchronization primitives. However, this is simple to implement using an integer counter and condition variable pair, as shown in Table 5.2.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>Integer, &gt; 0 for write, &lt; 0 for read</td>
</tr>
<tr>
<td>cond</td>
<td>Condition variable for signalling other threads</td>
</tr>
</tbody>
</table>

Table 5.2: IQ-barrier structure fields

A read lock is thus simply a decrement that must first ensure the counter is non-positive:

**Algorithm 5.1: iq_barrier_start_read**

Input: \( b \), an IQ-barrier structure  
Input: \( m \), a locked mutex

1. while \( b.state > 0 \) do
2. \[ \text{cond} \text{\_wait}(b.cond, m) \]
3. \( b.state := b.state - 1 \)

While the corresponding read unlock is an increment which signals any waiting threads if the counter has returned to zero:

**Algorithm 5.2: iq_barrier_finish_read**

Input: \( b \), an IQ-barrier structure

1. \( b.state := b.state + 1 \)
2. if \( b.state = 0 \) then
3. \[ \text{cond} \text{\_broadcast}(b.cond) \]
A write lock is implemented similarly, with the arithmetic and comparison operators swapped appropriately.

Using these operations as primitives, the multi-stage serialization for a stream is provided by an array of iq-barriers and a mutex. A stream uses only one serialization mutex, which is used with all iq-barriers and guarantees atomic transitions between stages. Queries use Algorithm 5.3, serialize_read, and insertions use the analogous serialize_write, passing the stage index as a parameter:

### Algorithm 5.3: serialize_read

**Input:** $s$: stage index  
**Data:** $B$: an array of $n_b$ iq-barriers, one per stage  
**Data:** $m$: a mutex

```plaintext
1 with locked m do
2   if $s > 0$ then (Release read lock for previous stage)
3     iq_barrier_finish_read($B[s-1]$)
4   if $s < n_b$ then (Acquire read lock for this stage)
5     iq_barrier_start_read($B[s], m$)
```

## 5.4 Algorithms

### 5.4.1 Initialization

VOLAP is designed to ingest data exclusively from clients in real-time. This introduces challenges for a new empty system since it is not possible to initially analyze the data set to determine an optimal data distribution. However, the range of the dimension hierarchies is known at system startup, which provides coarse bounds for the keys which may be inserted.

The system is initialized using the dimension hierarchy ranges as well as the initial number of workers and shards. To prepare for data ingestion, the manager retrieves this information from Zookeeper and derives the minimal box which encompasses all possible keys.

This box is then recursively split along each dimension in decreasing breadth order until a box has been produced for each initial shard. The resulting boxes are ordered such that boxes which were produced by splitting a larger box are adjacent. For example, if $B$ is split into $B_l$ and $B_r$, then the resulting array contains $[B_l, B_r]$. If $B_l$ is then further split into $B_{ll}$ and $B_{lr}$, then the resulting array contains $[B_{ll}, B_{lr}, B_r]$. A shard is created for each box, and
these are distributed among the initial workers in a round-robin fashion with replication.

The initial number of shards thus provides a degree of control over load balancing agility and skew tolerance. If the initial number of shards is much greater than the initial number of workers, then a given spatial region is initially distributed among many workers. For example, if it is likely that the majority of inserts will be within one quarter of the key space, choosing at least four times as many initial shards as workers will balance this load evenly.

Note, however, that this initial configuration is not fixed, skew will be adjusted for over time by the real-time load balancing scheme described in Section 5.5. An appropriately fine-grained initial distribution allows VOLAP to handle the initial loading of the system efficiently, and increases load balancing agility since smaller shards allow the load to be more easily balanced among workers without splitting.

5.4.2 Insertion

When a server receives an insertion request from a client, it is first processed by the receiver thread. This point is, by definition, the serial order of the request in the client’s operation stream. Accordingly, the receive thread first initiates the serialization process by reaching serialization point 1 as a write, that is, calling \texttt{serialize\_write}(1). Then, it forwards the request to a thread, which handles it with Algorithm 5.4, \texttt{server\_insert}.

\begin{algorithm}
\caption{\texttt{server\_insert}}
\textbf{Input:} $I$, insert message (point key and measure data)
\begin{algorithmic}[1]
\State \texttt{serialize\_write}(2)
\State \textbf{with read-locked} $l =$ the index lock \textbf{do}
\begin{algorithmic}[1]
\State Let $s =$ the ID of the shard to use, determined by inserting into the local index
\State Let $W =$ the set of all workers that host shard $s$
\end{algorithmic}
\State \texttt{serialize\_write}(3)
\State \textbf{foreach} $w \in W$ \textbf{do}
\begin{algorithmic}[1]
\State Enqueue insertion message $I$ for shard $s$ to worker $w$
\State Enqueue write commit mark (serialization step 4)
\State Wait for responses from all workers and record status
\State Respond to client with acknowledgement or error
\end{algorithmic}
\end{algorithmic}
\end{algorithm}

If boxes are expanded during insertion into the local index, the global image is not updated immediately, to avoid Zookeeper becoming a bottleneck. Instead, expanded boxes
are flagged, and the global image will be updated if necessary at the at the next configurable update interval.

For serializable execution, all messages to workers are enqueued for later transmission by the sender thread, and followed by a special message to mark the end of messages in the stream that correspond to this insertion. When the worker messages are received by the sender thread, they are forwarded to the appropriate worker. When the mark is received, all messages for this insertion have been sent in a correct order, so the sender thread calls `serialize_write(4)` which indicates the end of server serialization for this insertion.

Similarly to the server, these forwarded messages are first processed by the worker’s receive thread. Here, the first worker serialization point is reached, then the message is forwarded to a thread which handles it with Algorithm 5.5, `worker_insert`.

**Algorithm 5.5: worker_insert**

```
Input: I, insert message (point key and measure data)
Input: s, ID of shard for insertion

 serialize_write(7)
 with read-locked l = the shards lock do
   Let d = ∅  \hspace{1cm} (Shard data structure to use for insertion)
   if shard s has an insertion queue then (s is being split or received)
     d := the insertion queue for s
   else if shard s has been split then
     d := the “best” side according to the same criteria used by the server index
   else
     d := the data structure for s
   Insert the data item from I into d
 serialize_write(8)
 Respond to server
 Update Zookeeper statistics if necessary
```

The worker insertion algorithm itself is relatively straightforward, but a few things must be handled carefully to maintain correctness in the presence of real-time load-balancing (further discussed in Section 5.5).

First, a non-exclusive lock is taken when accessing the set of shards. This lock is used to provide atomicity when changing the set of shards, for example, when adding an insertion queue for migration. Note that insert and query operations only take a non-exclusive (read)
lock to avoid blocking other insertions or queries. An exclusive lock is only taken for a very short time when the set of shards or insertion queues must be modified to prepare or finalize a split or migration.

Due to the real-time nature of the system, the given shard ID may not always correspond to the same data structure. There are three cases: insert into an insertion queue if one is present, the appropriate side if the shard has been recently split, or otherwise the shard’s data structure directly (the typical case). In the insertion queue case, the insertion will later be applied to the appropriate shard data structure when the queue is drained after the load-balancing operation is complete.

Once the insertion is complete, the worker responds to the server, then updates statistics in Zookeeper if necessary. This is done in this order so that the processing of the insertion is not unnecessarily delayed by communication with Zookeeper.

5.4.3 Range Query

A range query on the server differs from an insertion since the query may need to be forwarded to many different shards. In Algorithm 5.6, server_range_query, the server first uses the index structure in its local image to determine which shards may be relevant to the query, that is, which shards have a bounding box which intersects the query box. It then forwards the resulting range queries to workers for processing.

Workers receive range query requests from servers, determine which shard data structures they must be applied to, and do so, in Algorithm 5.7, worker_range_query. Similar to the insertion case, the algorithm must consider both ongoing and completed splits, but rather than selecting a single structure, the query is applied to all applicable structures.
**Algorithm 5.6: server_range_query**

**Input:** \( q \), the query box

1. `serialize_read(7)`
2. \( \text{with read-locked } l = \text{the shards lock do} \) (Find shards and workers)
3. \( \text{Let } S = \text{the set of IDs of shards that intersect } Q \text{ (from local index)} \)
4. \( \text{Let } W = \text{an initially empty dictionary from worker IDs to sets of SIDs} \)
5. \( \text{foreach } s \in S \text{ do} \)
6. \( \text{Let } w = \text{the workers to query for shard } s \)
7. \( W[w] = W[w] \cup \{s\} \)
8. `serialize_read(8)`
9. \( \text{foreach } w, T \in W \text{ do} \) (Send messages)
10. \( \text{Enqueue range query/queries for shards } T \text{ to worker } w \)
11. \( \text{Enqueue read commit mark (serialization step 4)} \)
12. \( \text{Let } r = \text{an initially zero range query result} \)
13. \( \text{foreach } w, s \in W \text{ do} \) (Aggregate worker responses)
14. \( \text{Wait for response from worker } w, \text{ or skip if } w \text{ has died} \)
15. \( \text{Update } r \text{ with this result} \)
16. \( \text{Respond to requesting client with } r, \text{ or an error if too many workers died} \)

---

**Algorithm 5.7: worker_range_query**

**Input:** \( q \), the query box

**Input:** \( S \), a set of shard IDs to query

1. `serialize_read(2)`
2. \( \text{with read-locked } l = \text{the shards lock do} \)
3. \( \text{Let } r = 0 \)
4. \( \text{foreach } s \in S \text{ do} \) (Query shard(s))
5. \( \text{if there is a data structure } d \text{ for shard } s \text{ then} \)
6. \( r := r + \text{tree_query}(d, Q) \)
7. \( \text{else if shard } s \text{ has been split into sides } d' \text{ and } d'' \text{ then} \)
8. \( r := r + \text{tree_query}(d', Q) + \text{tree_query}(d'', Q) \)
9. \( \text{if shard } s \text{ has an insertion queue } q \text{ then} \)
10. \( r := r + \text{tree_query}(q, Q) \)
11. `serialize_read(3)`
12. \( \text{Respond to server with result } r \)
5.5 Load Balancing

Effective load balancing is crucial for scalable distributed systems. When the workload is unevenly partitioned among system resources, some portion of the system goes underutilized while the remainder struggles to pick up the slack. This has a negative impact on throughput, response time, and stability, which tends to get further compounded as the system scales up in size. However, the load balancing operations themselves can also incur significant costs due to the overhead of moving potentially large amounts of data over the network. A load balancing scheme which offers a good trade-off between effectiveness and overhead is required to maintain high throughput in a real-time environment.

VOLAP uses a real-time load balancing scheme which allows workers to be added, removed, or replaced dynamically in order to maintain performance in the face of changing system load. The operations performed during load balancing are carefully designed to not interrupt processing, so insertions and aggregate queries can continue to execute while load balancing is performed.

A separate background process called the manager initiates load balancing operations. The manager periodically analyzes the system state stored in Zookeeper and decides on suitable load balancing operations. It then initiates these operations, coordinating the necessary actions between workers and servers. For example, the manager may identify a worker that is overloaded and about to run out of memory. Based on this, it sends messages to workers instructing them to perform splits and/or migrations via the operations split_query, split, serialize_shard, and deserialize_shard discussed below. Note that the manager is a background process that only initiates operations which are then executed by the workers and servers. The manager process is therefore not a bottleneck for query performance, and it can reside anywhere in the system.

The split protocol and migration protocol in VOLAP have several steps which are punctuated by messages sent between nodes. Each step is designed to maintain availability so that incoming insert and aggregate query operations are not interrupted. The primary mechanism for achieving this is the creation of insertion queues where necessary, and atomically transitioning between states such that the insert and query algorithms always access only the appropriate data structures. In addition to allowing load balancing between existing workers, this scheme allows VOLAP to add, remove, or replace workers dynamically in order to maintain system performance as system load changes.
5.5.1 Shard Operations

The data in the system to be balanced is divided into several shards $D_i$ with corresponding bounding boxes $B_i$. In order to support load-balancing, the shard data structures provide the following operations:

- **split_query**($D_i, B_i$), which returns a hyperplane $h$ that partitions $D_i$ into $D_i^1$ and $D_i^2$ with bounding boxes $B_i^1$ and $B_i^2$, respectively, such that $D_i^1$ and $D_i^2$ are of approximately equal size. This operation does not perform an actual split or otherwise modify the data structure in any way.

- **split**($D_i, B_i, h$), which returns ($D_i^1, B_i^1, D_i^2, B_i^2$), where $D_i$ is partitioned into $D_i^1$ and $D_i^2$ with bounding boxes $B_i^1$ and $B_i^2$, respectively, such that $D_i^1$ and $D_i^2$ are spatially separated by hyperplane $h$. This operation requires $D_i$ to remain in a consistent state for its duration, that is, new items may not be inserted into $D_i$ while the split is in progress.

- **serialize_shard()**, which returns a flat binary blob $b$ containing the data in $D_i$ suitable for network transmission, and the corresponding **deserialize_shard**(b), which builds the data structure from such a blob. Like **split**, this operation requires $D_i$ to remain in a consistent state while it is being serialized.

The **split_query**, **split**, **serialize_shard**, and **deserialize_shard** operations themselves are omitted here for brevity. The more interesting aspects of load balancing are related to maintaining their invariants while continuing to process insert and query operations. For example, when **serialize_shard** is being used during a migration, new items may not be inserted into the data structure, though the worker may still need to process insertions for that data structure immediately. The remainder of this section describes how VOLAP handles such situations.

5.5.2 Processing Requests During Load-Balancing

Correctly splitting or migrating a shard requires a consistent snapshot of the shard for a particular point in time. However, a real-time system requires that operations can be processed at any time, including when shards are being split or migrated. Though the Hilbert PDC-tree is capable of handling insertions and queries concurrently, it would not be correct
to simply insert into the data structure while serializing, since it would not be clear which
insertions have been included in the serialized shard and which have been missed. In order
to maintain consistency, the system must ensure at a higher level that each insert ultimately
results in a single data item being present in the correct shard data structure.

To handle this situation, workers create an *insertion queue* for any shard that is being
split or received in a migration. During the operation, insertions for the shard are inserted
into the queue rather than the shard itself, as illustrated in Figure 5.7. Queries are directed to
both the shard and the insertion queue to ensure results are up to date. The insertion queue
uses the same data structure as shards, allowing insertions and queries to be processed by
the queue with the same performance as shards themselves.

![Diagram showing insertion queue during migration and split](image)

(a) Insertion queue during a migration  (b) Insertion queues during a split

Figure 5.7: Distribution of insertion and query requests to a shard $D_i$ and its insertion
queue(s) during a split or migration.

For migrations, when the shard transfer is complete, the insertion queue on the destination
worker is drained into the new shard data structure. For splits, when the operation is finished,
the insertion queue is drained into the two resulting shards using the same rules used by the
server index. After being drained, the insertion queue is destroyed and normal use of the
shard data structure resumes.

The implementations of the algorithms for each stage of the split and migration protocols
follow directly from the below descriptions. However, one detail is particularly important:
whenever the set of shards or insertion queues on a worker are modified, an *exclusive* shards
lock is held. This is the same lock that is held non-exclusively by the worker algorithms in
Section 5.4, ensuring these algorithms always have a consistent view of the set of shards
on the worker. Thus, strictly speaking, insert and query processing is occasionally blocked
during the split and migration processes. However, this lock is only held exclusively for a very short time in order to update pointers in small dictionaries, and never while executing expensive operations like tree splitting or serialization.

### 5.5.3 Shard Migration

Shards can be transparently moved from one worker to another while the system continues to service requests. This is useful for adding new workers to elastically scale the system as the data size increases, as well as balancing the data load among the existing workers when necessary.

The basic strategy for maintaining responsiveness during migration is to direct insertions to the destination worker and queries to both it and the source worker while the shard is migrating. Once the destination worker receives the shard from source worker, it is activated there and removed from the source worker. The migration process consists of several stages, each associated with a message delivery as illustrated in Figure 5.8. The process is coordinated by the manager, and a step only begins once the previous step is complete:

1. **Manager ⇒ Destination Worker: MigratePrepDest(W_s, s)**
   To begin a migration, the manager notifies the destination worker that it will be receiving a new shard $s$. The destination worker creates an insertion queue for $s$ to handle any insertions that may arrive during the migration.

2. **Manager ⇒ Servers: MigratePrepServ(W_s, W_d, s)**
   Once the destination worker is ready, the manager notifies all servers that shard $s$ is about to migrate from $W_s$ to $W_d$. The servers begin sending insertions for $s$ to the destination worker, and queries for $s$ to both the source and destination worker. To ensure correct serialization, the server does not respond to this message until all pending insertions for $s$ that were sent to $W_s$ are finished.

3. **Manager ⇒ Source Worker: MigrateBegin(W_d, s)**
   Once all servers are aware of the pending migration and no insertions for $s$ are enqueued to the source worker, the manager instructs the source worker to send shard $s$ to $W_d$.

4. **Source Worker ⇒ Destination Worker: MigrateData(s, W_d, b)**
   The source worker serializes shard $s$ into a blob $b$ and sends it to $W_d$. 
5. **Destination Worker ⇒ Manager: MigrateDone(s)**
   
   When the destination worker receives shard $s$, it adds it to its set of data structures, drains and destroys the corresponding insertion queue, then notifies the manager that it has received the shard.

6. **Manager ⇒ All Servers: MigrateDone(s)**
   
   When the migration is complete, the manager notifies all servers, which cease sending messages for $s$ to the source worker.

7. **Manager ⇒ Source Worker: DeleteTree(s)**
   
   Now that $s$ is no longer considered to reside at the source worker by servers, the manager instructs the source worker to delete it.

8. **Manager ⇒ Zookeeper:**
   
   Finally, the manager updates Zookeeper to reflect the new location of $s$, and the migration is complete.

![Migration process](image)

**Figure 5.8: Migration process**

### 5.5.4 Shard Splitting

When a shard becomes too large, it may need to be split into two smaller shards in order to maintain balance among workers. Like migration, the process of splitting a shard is initiated and coordinated by the manager, and the protocol is designed such that insert and
aggregate query processing can proceed normally while shards are being split. The split process, illustrated in Figure 5.9, has fewer stages than migration since workers do not need to coordinate with each other:

1. **Manager ⇒ Worker: SplitQuery(s):**
   First, the manager arbitrarily selects one of the workers replicating the shard and requests that it return a hyperplane with which to split the shard.

2. **Manager ⇒ Worker: SplitRequest(s)**
   Once the manager receives the split hyperplane, it sends a split request with the hyperplane to all workers replicating the shard. The worker creates an insertion queue for shard s, then begins the split process.

3. **Worker ⇒ Manager: SplitComplete(s, s’, s’’)***
   When a replica split completes, the worker notifies the manager that s is split into s’ and s’’.

4. **Manager ⇒ All Servers: SplitUpdate(s, s’, s’’)***
   Once the manager receives the SplitComplete message from each worker, it notifies all servers that s has been split into s’ and s’’. The servers begin sending insertions and queries to s’ and s’’.

5. Finally, the manager updates Zookeeper to remove the old shard s and add the two new generated shards s’ and s’’.
5.5.5 Replication

If a worker node exceeds a configurable timeout, it is assumed to have died and its shard replicas are removed from the system image. The load balancer, which periodically checks the system image for any shards that have less than $N_r$ copies, will then initiate a replication operation of the shard to another worker. The replication process is essentially a migration, except the shard is not deleted from the source worker. Once a copy of the shard has been sent and activated on the destination worker, the manager updates Zookeeper to reflect that the destination worker now replicates the shard.

5.5.6 Load Balancing Algorithm

VOLAP has a modular design that allows for various optimizers which implement load balancing algorithms. The manager provides access to the system state as stored in Zookeeper, and periodically requests load balancing operations from the optimizer. If the optimizer returns a set of operations, the manager initiates them and, once all operations are complete, resumes periodically requesting load balancing operations from the optimizer.

The experiments below use an optimizer that balances the memory load of workers, so if a single worker becomes too full, data will be migrated to a worker with more free memory. The improvement $I$ for a migration of a shard $s$ from worker $W_s$ to $W_d$ is the difference between the current imbalance of the workers and the imbalance if the migration were to be
performed. Denoting the size of a worker or shard $X$ as $\overline{X}$:

$$I = |\overline{W}_s - \overline{W}_d| - |(\overline{W}_s - \overline{s}) - (\overline{W}_d + \overline{s})|$$

(5.1)

For example, if $\overline{W}_s = 1000$, $\overline{W}_d = 500$, and $\overline{s} = 250$, then

$$I = |1000 - 500| - |(1000 - 250) - (500 + 250)|$$

$$= |500| - |750 - 750|$$

$$= 500$$

In this case, migrating $s$ results in the two workers being completely balanced, so the maximum possible improvement, their original difference in size, is achieved.

The algorithm proceeds by evaluating every possible migration and ranking them by improvement. In order to avoid migrations that do not result in a significant enough improvement to justify their expense, the user can specify a migration threshold parameter $T_m$. For a migration to be considered, the improvement must exceed the shard size scaled by the migration threshold, that is:

$$I \geq T_m \cdot \overline{s}$$

(5.2)

Then, potential splits are considered by evaluating the migrations they would enable. For each shard, the optimizer considers the migrations that would be possible if the shard were split in half, and adds the split to the list of potential operations using the same criteria as migrations.

The optimizer then chooses operations in order of decreasing improvement. To prevent a worker from being too heavily loaded by load balancing operations, at most one migration to or from a given worker, and at most one split on a given worker, is chosen in a single optimization round. When an operation is chosen, the relevant workers are flagged, and subsequent potential operations that involve those workers are skipped. Any such skipped operations will likely be chosen in the next optimization round if they remain worthwhile.

5.6 Correctness
Theorem 5.1 VOLAP serializable execution

With serializable execution enabled, an insertion in a client’s operation stream will be reflected in the result of any subsequent query in that client’s stream.

Proof. Let $I$ be some insertion, and $Q$ be any subsequent query in the same client’s operation stream whose bounding box contains $I$. When $I$ is received by the server, it is processed by Algorithm 5.4, server_insert, which uses the local index to determine the shard ID $s$ which will be used for this insertion, and the set of corresponding workers $W$. A read lock is held during this selection to ensure that it is performed on a consistent snapshot and not interfered with by load balancing operations. Serialization points 2 and 3 ensure that the corresponding box in the local index is expanded if necessary before another thread processing a query can proceed to search the index, and thus $s$ will be included when processing $Q$.

The insertion is then forwarded to workers by the send thread, where serialization point 5 ensures that $I$ is sent to workers before $Q$. Once received by the worker, the insertion is handled by Algorithm 5.5, worker_insert.

There are several cases to consider, noting that transition between these cases is guaranteed to be atomic through the use of the server’s index lock and the worker’s shards lock:

1. In the usual and simplest case, shard $s$ is neither split nor migrated at any time during the insertion. Then, $I$ is received by the worker and inserted directly into the data structure for $s$. Worker serialization points 7 and 8 ensure that this is complete before another thread which receives $Q$ can proceed to find and query the same data structure. $Q$ will thus include $I$ in its result.

2. If the shard $s$ is split, there are three possible cases:

   (a) In the typical case, the shard is split after the insertion is complete. Then, a server processing $Q$ will not choose $s$ from its index at all, but instead choose the resulting shard(s) that intersect the query box. At least one such shard contains $I$ and therefore will be included in the query since its bounding box will intersect the query box. The same argument applies if these sides are further split any number of times: the server index will always contain a shard whose bounding
box contains $I$, and thus some shard containing $I$ will always be queried while processing $Q$.

(b) The shard is in the process of being split. In this case, an insertion queue for $s$ was created at the start of the split process. The worker receives $Q$ with shard ID $s$, sees that $s$ has an insertion queue, and queries both the original shard data structure and the insertion queue. $I$ will therefore be included in the result regardless of whether it was processed before the split began or while the split was in progress.

(c) The shard has been split on the worker, but the split has not yet been finalized and broadcast to the server. In this case, when the worker receives $Q$ for shard $s$, it discovers using its split history table that $s$ has been split into $s'$ and $s''$ and queries both sides if their bounding boxes intersect the query box. Thus, $I$ will be included in the query result.

3. Similarly, if the shard $s$ migrates, there are two cases to consider:

(a) The shard is migrated completely before $Q$ is received. In this case, the server is aware of the new location for the shard, and simply forwards the query to the appropriate worker which will process it as usual.

(b) The shard is being migrated when $I$ is received. The first two steps in a migration cause all servers to be aware of the pending migration, and for insertions to be sent to the destination worker. In this case, $I$ is first inserted into the insertion queue at the destination worker. If $Q$ arrives before the migration is complete, then the insertion queue will be queried as well, and thus $I$ will appear in the result. Otherwise, if $Q$ arrives after the migration is complete, $I$ has been drained into the data structure for shard $s$ and will thus be queried normally.

Therefore, in all possible cases, any insertion $I$ will be included in the result for a subsequent query $Q$ whose bounding box contains $I$. 

\[ \square \]
5.7 Implementation Details

VOLAP is implemented in approximately 23 thousand lines\(^1\) of C++, using the standard library where possible, and a few POSIX facilities and boost libraries as necessary.

The server, worker, and manager are implemented as independent programs which take a port number as a parameter. Thus, it is possible to run several components on a single machine, allowing for flexible deployment such as running the manager on the same node as a server, or testing on a single machine.

All parameters are managed with a unified configuration system. Parameters can be set from the command line, a configuration file, or Zookeeper. Zookeeper support allows the system configuration to be easily used by all nodes without the complication of an additional distributed system for managing command line arguments or configuration files. For example, if a new worker is launched to balance the load, it can discover all necessary information about the system, including its own configuration parameters, from Zookeeper.

Various clients are used for benchmarking and testing. The experimental figures in the following section are derived from the log output of running several loading and benchmarking clients, which log numerous details about their performance, such as their mean, minimum, and maximum throughput and latency. System-wide statistics, such as the amount of data stored on workers, are periodically logged by the manager which has access to this information via Zookeeper.

A client to test the correctness of serializable execution sends a stream of interspersed insertions and queries constructed such that each query result is predictably influenced by the previous insertions. Each query result is verified to have the expected value, ensuring that serializable execution is implemented correctly, that is, that each query result contains exactly all insertions issued previously in the stream, and none later. This client has never been observed to fail over many thousands of tests on various system configurations.

Data is loaded via an abstract interface with two implementations. One generates synthetic data with a Zipf distribution according to a user-specified skew parameter using the methods described in Appendix B. The other loads data from delimited text files, which can be used to load data generated by the TPC-DS test suite, or any other typical delimited text format such as CSV or TSV. Dimension hierarchies themselves are loaded from other files in the same format.

\(^1\) As measured by David A. Wheeler's `sloccount`
5.8 Experimental Evaluation

The performance of VOLAP is evaluated with respect to the system parameters shown in Table 5.1, the query coverage, and the workload mix.

**Definition 5.1 Workload Mix**

The workload mix, or simply workload, is the percentage of an operation stream that consists of insertions. For example, a stream with a workload mix of 25% contains 25% insertions and 75% aggregate queries.

Except where otherwise noted, the TPC-DS [63] decision support data set from the Transaction Processing Council is used, with \( d = 8 \) hierarchical dimensions as shown in Figure 2.2. Experiments were performed on Amazon EC2, using c3.8xlarge instances for servers, c3.4xlarge instances for workers, and c3.2xlarge instances for clients and the manager. These instances use 32, 16, and 8 cores of an Intel Xeon E5-2680v2, respectively. All instances are running Amazon Linux with Linux 3.14.35, ZeroMQ 4.0.5 and Zookeeper 3.4.6.

**5.8.1 High-Velocity Data Ingestion**

Figure 5.10 shows the performance of VOLAP for high-velocity data ingestion up to a database size of \( N_i = 1 \) billion elements, using synthetic Zipf-distributed data. The system has 20 workers and 4 servers, and 4 clients, one per server. Each client sends an asynchronous stream of bulk insertions, the value shown is the sum of the throughput of all clients. In this non-elastic test with a purely bulk ingestion workload, VOLAP ingests approximately 1.25 million items per second consistently throughout the experiment.

Note that the following tests that use individual insertions and mixed operation streams show much lower throughput because bulk insertions are handled specially by VOLAP and can achieve much higher throughput due to the reduced overhead per item.
Figure 5.10: Data ingestion performance as database size $N_i$ increases. Zipf data, $s = 1.0$, $p = 20$.

5.8.2 Serializable Execution Impact

The performance impact of serializable execution is shown in Figure 5.11. The socket configurations refer to the different architectures described in Section 5.3.4. All three experiments use the same TPC-DS data and set of queries, and are run for 10 seconds. The effects of serializable execution can be seen by comparing “serializable” with “single socket”, which both have the same socket configuration. The impact is greater when there is a mix of insertions and queries, as expected, since synchronization is required only when an insertion is followed by a query, or vice-versa. In the worst case, at a 50% mix, serializability comes at a cost of approximately 20 thousand operations per second. A significant cost, but one that only affects the client with serializable execution enabled. Note that a 50% mix here results in a stream that is often an insertion followed by a query, then another insertion, and so on; the worst-case scenario for serializable execution overhead.
Figure 5.11: Performance impact of serializable execution. “Thread sockets” has server threads communicate with workers directly using a thread-local socket. “Single socket” has all threads send to workers via a single socket, and “Serializable” uses the same architecture but with added synchronization to guarantee serializable execution.

5.8.3 Real-Time Load Balancing

The real-time load balancer in VOLAP mediates the elasticity of the system. As workers are added to VOLAP, the load balancer automatically moves data items to the new workers to balance the workload. Figure 5.12 shows the impact of load balancing during a horizontal scale-up experiment. In this experiment, load phases are interleaved with mixed insertion and query phases. At the start of each load phase, two additional workers are added to account for the increase in database size. The red region shows the minimum and maximum number of data elements stored on a worker, with the darker red line showing the mean. Similarly, the green region and line show the size of shards.

When new workers are introduced they are empty, causing the minimum to go to zero. The effects of the load balancer are clearly visible as the gap between minimum and maximum worker size is reduced by moving data to the newly introduced workers. The number of migration operations is shown as a dotted purple line associated with the right y-axis. Once balance is achieved, loading proceeds, increasing the minimum and maximum size per worker as new elements are inserted.

Note that VOLAP can load balance at any time, including while data is being ingested and queries are being performed. This experiment proceeds in phases in order to demonstrate load balancing and ensure a stable benchmarking environment for evaluating scale-up performance. In general, load balancing is performed concurrently with insertions and
aggregate queries when necessary, as determined by the manager.

![Graph showing the number of elements per worker with increasing system size](image)

Figure 5.12: Number of elements per worker with increasing system size: $N_i \approx p \cdot 500$ million, $p = 4...20$, $m = 2$

### 5.8.4 Horizontal Scale-Up Performance

Figure 5.13 shows the insert and aggregate query performance for various workloads as the system size increases. This data is from the same experiment as shown in Figure 5.12, where two new empty workers are added at each scale-up step, demonstrating the elastic capabilities of VOLAP. For each system size with $p$ workers and $N_i \approx p \times 50$ million data elements, measurements are performed for insertions as well as queries with low (below 33%), medium (between 33% and 66%), and high (above 66%) coverage. Each workload mix is measured for 5 seconds. The throughput and corresponding latency are shown in Figure 5.13.

Figure 5.13 shows that VOLAP scales well in an elastic cloud environment. As the database size increases and processing resources (workers) are added in a fixed ratio, VOLAP maintains its performance over the entire range of database sizes. Once the system is loaded with a few million elements, the insertion curves in Figure 5.13 are relatively flat, at approximately 50 thousand insertions per second with a latency of 0.05 seconds.

Queries show similar performance despite the wide range of coverage due to the Hilbert PDC-tree data structure’s support for large aggregations. As expected, queries are more impacted by the increasing database size, but still exhibit a gentle and predictable slope. If necessary, this could be compensated for by adding workers slightly more aggressively to account for the overhead of the increased system size.
Figure 5.13: Query and insert performance with increasing system size: $p = N_t / 50$ million ($4 \leq p \leq 20$)

5.8.5 Insert and Query Performance

Figure 5.14 shows the throughput and latency for inserts and queries at a fixed database size of 1 billion items with various workload mixes and query coverages.

Workload mix has a significant impact on throughput because the time spent for insertions and aggregate queries may vary considerably, and insertions may trigger changes to a shard data structure that is concurrently being used to answer queries. Coverage has a significant impact on performance because it globally impacts the number of different trees and workers that must be searched to answer an aggregate query, and locally impacts the number of nodes in each tree that must be searched.

Figure 5.14 again demonstrates the “coverage resilience” of the Hilbert PDC-tree: query performance is nearly identical regardless of coverage. This result would be counter-intuitive for a typical tree, but the Hilbert PDC-tree includes cached aggregate values in all tree nodes which speeds up aggregation. Figure 5.14 shows that this behaviour is reflected in the overall performance of VOLAP.

In the experiments in Figure 5.14, insertions were approximately three times faster than queries, with a predictable linear relationship between workload mix and overall performance. This shows that insertions do not considerably impact the performance of concurrent queries, due to the minimal locking employed by VOLAP and the Hilbert PDC-tree.
Figure 5.14: Performance for various workload mixes and query coverages: \( N_t = 1 \text{ billion}, p = 20, m = 2 \)

### 5.8.6 Coverage Impact

A more detailed analysis of the impact of query coverage on performance is shown in Figure 5.15. Both the impact on individual query time and the number of shards searched are shown as a heat map.

Figure 5.15a shows that the majority of queries are executed very quickly, with a few outlying times for queries with very low coverage. This reflects the aggregation behaviour of the Hilbert PDC-tree: with high coverage it is likely that suitable cached aggregate values will be found at higher levels in the tree, making deeper traversal unnecessary. However, with low coverage, high level directory nodes are less likely to completely contain the query region, so it may be necessary to walk to the leaf level several times to find individual values.

Figure 5.15b shows that the relationship between query coverage and the number of shards searched is approximately linear, where increasing coverage requires an increasing number of shards to be searched. However, there are some outlying points at around 50% coverage where significantly more shards must be searched. This is due to queries that overlap the boundaries of several shards, requiring a larger number of shards to be queried.
Figure 5.15: Effect of coverage on query performance. $N_i = 1 \text{ billion}; p = 20.$

5.8.7 Skew Tolerance

Figure 5.16 shows the performance of VOLAP on Zipf-distributed data with increasing skew. The system is configured to maintain ingestion performance, with 80 initial shards, a minimum balance improvement of 2 million, and worker statistics updated every second. Query times shown are the average for all coverages at the maximum database size of 1 billion items.

Figure 5.16: Performance with increasing skew; $N_i = 1 \text{ billion}; p = 20.$

Because of the multi-dimensional nature of the data and the distribution algorithm described in Section 5.4.1, skew and performance do not have a straightforward linear relationship, but as Figure 5.17 shows, more load-balancing operations are necessary to maintain performance as skew increases. Splits tend to impact performance most significantly, since they only occur when migration of existing shards can not address the imbalance. Worker
threads are thus kept busy for longer before the migration can occur and balance be improved. This is particularly visible in this experiment with a skew of 3.0, where ingestion performance drops to just above 800 thousand inserts per second. However, in general, VOLAP maintains relatively stable performance over a wide range of skew, sustaining well over a million inserts per second on average. This is because once heavily loaded shards are split and/or migrated, their load is more evenly distributed among workers, so the skew of the data contributes less to skew between workers as time progresses.

![Graph showing load balancing behaviour with increasing skew](image)

Figure 5.17: Load balancing behaviour with increasing skew; \( N_i = 1 \) billion; \( p = 20 \).

The effect of load balancing over time for a run with an extremely high skew of 8.0 can be seen in Figure 5.18 which shows the number of items stored on each worker over time. While the load between workers becomes highly skewed at times early in the experiment, the effect of the load balancer is clearly visible as the balance is restored to be relatively uniform.

![Graph showing distribution of data across workers](image)

Figure 5.18: Distribution of data across workers during a run with skew 8.0
5.8.8 Replication Impact

Figure 5.19 shows the operation latency with varying values of \( N_r \), \( W \) and \( R \), with \( N_i = 300 \) million items. The number of worker nodes is the same regardless of the value of \( N_r \), and consequently the amount of load on the set of workers is proportional to \( N_r \). As a result, increasing \( N_r \) with this configuration has a negative impact on performance.

The addition of replicas increases latency by approximately 0.1 to 0.2 seconds in this configuration, as shown in Figure 5.19. Since all of the compute nodes in this experiment communicate over a low latency local network, there is little variance in the time it takes each worker to return a response to a server. Because of this, the value of the read and write quorum has a minimal effect on operation latency with this configuration.

![Figure 5.19: Latency for a 50% workload mix with various replication factors; \( N_i = 300 \) million, \( p = 20 \), \( m = 2 \).]

5.9 Summary

VelocityOLAP (VOLAP) is a scalable OLAP system which allows high-velocity data to be queried in real-time.

VOLAP makes use of dimension hierarchies, is highly scalable, exploits both multi-core and multi-processor parallelism, and can guarantee serializable execution within a client operation stream. VOLAP is a fully decentralized and elastic system which supports horizontal scaling on commodity hardware or modest cloud instances.

VOLAP uses the Hilbert PDC-tree as an underlying data structure, and extends its properties of high ingestion throughput and efficient support for large aggregate queries to a
distributed context, supporting databases that are too large to host on a single machine.

Experiments show that VOLAP performs and scales well with a variety of data sets, and can dynamically adapt to changing load or network architecture while maintaining performance suitable for real-time use.
Chapter 6
Eventual Consistency and Staleness

6.1 Introduction

The previous chapter described VOLAP, which can provide serializable execution for a single client’s operation stream. Within such a stream, there is no concept of query staleness, since operations have a strict ordering, and query results always include all relevant insertions encountered earlier in the same stream.

However, between multiple client streams, VOLAP can’t provide this guarantee. Instead, eventual consistency is provided, which is theoretically much weaker but commonly used in distributed systems. In this context, staleness is an important attribute of the system. For example, users may wish to be confident that queries include any relevant data inserted at most three seconds ago by any other client. This is especially important because the system configuration presents a fundamental trade-off between operation latency and data consistency. Most significantly, shard replication, and partial quorum replication in particular, greatly affect this trade-off.

Redundant replication in a system introduces the problem of consistency. Since networks are unpredictable, each insert operation will arrive at the $N_r$ different replicas at different times. This leads to the data stored in replicas being inconsistent from one another until each insert is applied to each of the $N_r$ replicas. Thus, a query can return an outdated result, and the staleness of a query depends on which replicas were used to calculate the result.

One widely popular method of approaching this problem is to implement a quorum consensus algorithm [41, 32, 33]. With a quorum consensus, each write operation is considered complete only after $W$ replica replies confirming a successful write have been received. Similarly, each read operation requires a result from $R$ replicas before a final result is returned. $W$ and $R$, or the write quorum and read quorum, are system parameters.

---

1. Note that staleness literature often denotes the number of replicas as simply $N$, but $N_r$ is used here to avoid ambiguity with the number of items $N_i$. 

---

This chapter is a more detailed description of previously published work [15].
used to ensure each write or read operation has been applied to a subset of replicas. Setting \( W + R > N_r \) guarantees consistency because each read will contain at least one of the \( W \) most recent writes. In order to decrease latency, many distributed data stores [23, 53, 5] use partial quorums, i.e. \( W + R < N_r \), thereby accepting the possibility of data staleness. For example, the default settings for Apache Cassandra [53] are \( N_r = 3 \) and \( R = W = 1 \).

Despite the weak theoretical guarantee of eventual consistency which allows results to be arbitrarily stale, many quorum-replicated data stores are in use that provide only eventual consistency with partial quorums. Probabilistically Bounded Staleness (PBS) [4, 5] provides an explanation for the experience by practitioners that eventual consistency in replicated data stores with partial quorums is often “good enough” in practice.

However, PBS is targeted at key-value data stores where queries require only a single value, and does not account for systems like VOLAP which focus on aggregation. This chapter studies data consistency in the context of quorum-replicated distributed OLAP systems and presents Aggregate Probabilistically Bounded Staleness (A-PBS), a measure for examining consistency for aggregate queries. Whereas the PBS measure for data stores refers to the staleness of a single data item, the more complex A-PBS measure described here is determined by the staleness of the entire set of aggregated data items, possibly the entire database. Furthermore, for aggregate queries, staleness can be determined either by the number of missed data items or by the error observed for the result of the aggregation. These issues do not arise for PBS in data stores.

A-PBS provides a formal model for analyzing eventual consistency which accounts for these inherent issues with aggregate queries in a quorum-replicated environment. In addition, a system model and Monte Carlo simulation for estimating eventual consistency for aggregate queries is provided. Given a list of simulation parameters which reflect the performance of a system, this model and simulation can be used to easily explore the trade-offs between consistency and latency in quorum-replicated distributed OLAP systems without the need for expensive experimental testing.

Experiments with VOLAP show that the staleness of aggregate queries predicted through A-PBS and Monte Carlo simulation is very close to that observed in the real system. For example, the difference between the simulated and observed average number of missed data items in an aggregate query covering 100,000 data items was 0.5. A-PBS analysis also confirms observations with VOLAP that a partial quorum with \( (N_r=3, W=0, R=1) \) works
well in practice. Even very large aggregate queries that cover the entire database which are issued immediately after the last insert have ≈ 40% probability of zero staleness. If staleness occurs for such aggregate queries, the expected number of missed data items is only 0.8, resulting in only a very small numerical error in the aggregate query result for \( \sum \) aggregation, and very close to zero probability of any numerical error for \( \max \) aggregation.

In summary, though eventual consistency provides only a weak theoretical guarantee, in practice quorum-replicated systems that provide only eventual consistency can provide sufficiently fresh query results for real-time use. A-PBS provides a theoretical framework for analyzing the behaviour of such systems, and experiments with VOLAP demonstrate that this framework accurately reflects observed behaviour. Though used here to analyze the behaviour of VOLAP, the A-PBS measure itself, as well as the corresponding model and simulation, are general and may be applied to any suitable system.

### 6.2 Related Work

Several works have been written on the topic of imprecise or ambiguous data in OLAP systems [14, 19, 59], where dimensional data and measures which are uncertain by nature are explored, rather than uncertain but eventually consistent data caused by a lack of synchronization within a distributed system. For example, all measures may have a certain amount of error with a known distribution. Related work [14] proposes a modified OLAP model which incorporates this concept of uncertain data and presents different metrics of query correctness within this context. However, since these metrics are defined within the context of inherently uncertain data, the model and corresponding metrics are inapplicable to eventually consistent OLAP systems like VOLAP.

A different approach [66] introduces middle-ware for distributed OLAP systems which manages replication, insert, and query operations to guarantee a certain freshness bound. A normalized “freshness index” is described, which measures how consistent a replica of a set of data is at the current point in time. Here, a model is used where query-answering OLAP nodes receive batch updates from writable OLTP nodes. The “delay freshness” is used to compute the freshness index: the time of the last update of a node \( \tau(c) \), divided by the commit time of the most recent transaction on an OLTP node \( \tau(c_0) \).

The A-PBS measure presented in this chapter is rooted in Probabilistically Bounded
Staleness (PBS) for key-value stores [4, 5], which examines eventual consistency in the context of quorum-based key-value stores. PBS provides a probabilistic metric of staleness for simple partial quorum \((W + R \leq N_r)\) read operations. PBS \((k, t)\)-staleness is defined as a property of a key-value quorum system which, with a specified probability, returns a result within \(k\) versions of the most recent write for reads sent \(t\) units of time after a write response has been received by at least \(W\) replicas for each write relevant to the read. A model and corresponding simulation for evaluating PBS \((k, t)\)-staleness have also been presented for PBS by the same authors [6]. In it, distributions describing the time taken to read and write keys, as well as send read and write response messages, are sampled to estimate a system’s probability of staleness.

6.3 Aggregate Probabilistically Bounded Staleness (A-PBS)

*Aggregate Probabilistically Bounded Staleness* (A-PBS) is a means of analyzing consistency in distributed OLAP systems. Like PBS, A-PBS defines metrics examining the consistency of aggregate queries in terms of missed writes. Unlike PBS, new consistency metrics are introduced that view consistency from the perspective of numerical error, since aggregate queries, particularly in OLAP systems, are conventionally numerical by nature [40].

6.3.1 Data Streams and Queries

While queries in key-value stores retrieve a single value specified by a key from a node, aggregate queries may involve a much larger percentage of the data. For example, a single query in a typical aggregate OLAP system can operate on anything from a single value to all values across all nodes in the system. This is an important distinction which divides key-value staleness analysis from aggregate OLAP-style staleness analysis.

In order to analyze such systems, it is necessarily to characterize the operations that define their interaction with a user. For this purpose, the previously described *coverage* (Definition 3.7) is used for queries, and the input *data stream* describes the write history of the system.
**Definition 6.1** *Data stream* \( \text{DATA}(n, f, D) \)

An input data stream \( \text{DATA}(n, f, D) \) is a stream of \( n \) insert operations, where each insert has a measure value sampled from the distribution \( D \) and is sent from a client every \( i = 1/f \) units of time.

Using these two definitions, a sequence of insertions followed by a single query can be modelled. For example, the consistency under maximum load of a system which can process at most 10000 insertions per second and contains 1 000 000 items can be analyzed by modelling a \( \text{DATA}(1\,000\,000, 10000, D) \) stream, and considering the result of a query initiated some time after all operations in the data stream have been sent. Figure 6.1 shows a graphical representation of a simple data stream.

![Graphical representation of a simple data stream](image)

**Figure 6.1**: A \( \text{DATA}(5, f, D) \) insert stream. White circles represent points in time at which inserts in the stream are sent from the client. All inserts are spaced \( i = 1/f \) seconds apart.

### 6.3.2 \((t,k)\)-staleness

Based on the above definitions, it is possible to define the first A-PBS metric of staleness:

**Definition 6.2** \((t,k)\)-staleness

Given a \( \text{DATA}(n, f, D) \) stream, a query \( Q \) initiated \( t \) units of time after the last insert in the stream has \((t,k)\)-staleness if and only if more than \( k \) insert operations covered by the query’s bounding box were not included in the result of \( Q \).

Note that a query having \((t,k)\)-staleness does not necessarily imply that the query result is incorrect. For example, the result of a \text{max} aggregation is likely to be unaffected by missing a few data points, that is, the query may have \((t,k)\)-staleness but still return a correct result. A \text{count} aggregation, however, will always return an incorrect result if any points within the query’s bounding box are missed.
Essentially, for a given data stream and query, \((t, k)\)-staleness makes a binary “fresh” or “not fresh” judgment of that particular query and stream. If the query happens to miss more than \(k\) relevant points in its aggregate result, it has \((t, k)\)-staleness.

\((t, k)\)-staleness has two important “slack” parameters, \(t\) and \(k\). \(t\) refers to the units of time between the initiation of the last insert in the stream and the initiation of the query. Since eventually consistent systems always converge towards consistency over time, greater values of \(t\) will lead to fewer queries with \((t, k)\)-staleness, while smaller values of \(t\) will be more likely to yield queries with \((t, k)\)-staleness. \(k\) corresponds to the maximum number of allowable missed inserts; the smaller \(k\) is, the more likely queries are to have \((t, k)\)-staleness.

Figure 6.2 provides a visual representation of the slack parameters in \((t, k)\)-staleness. As time proceeds from left to right, insert operations, represented by white circles, are sent to the system at a rate of \(i = 1/f\) units of time. Once the final insert has been initiated, \(t\) units of time pass until the query, represented by the black diamond, is initiated.

![Diagram](image)

Figure 6.2: An insert stream and query for \((t, k)\)-staleness. White circles and black diamond represent the respective initiation time of the inserts and query.

Since aggregate query freshness depends on each point in a data stream, ensuring that queries don’t have \((t, k)\)-staleness is more difficult compared to the key-value PBS equivalent. This is clearest when considering \((t, k=1)\)-staleness: in key-value systems, there is only one point of interest which can be stale, but in an aggregate system, many points relevant to a query can be stale, and any of these points being stale can give the query \((t, k=1)\)-staleness. Figures 6.3 and 6.4 illustrate queries with and without \((t, k=1)\)-staleness, respectively. In these figures, the top axis represents the time a query or insert is initiated, while the bottom axis represents the time at which the corresponding insert is readable, or the time at which the query begins.
Figure 6.3: A query that has \((t,k=1)\)-staleness. The last two inserts in the stream and the query are reordered, so more than \(k = 1\) inserts are missed.

Figure 6.4: A query that does not have \((t,k=1)\)-staleness. Since \(k = 1\), the reordering of the last insert in the stream and the query does not impact \((t,k)\)-staleness.

### 6.3.3 Data Stream Sizes

From the perspective of \((t,k)\)-staleness, the \(f\) parameter from a \(\text{DATA}(n, f, D)\) data stream also partially determines the effective number of inserts covered by the query. Since each insert is initiated \(i = 1/f\) units of time before the next, the initiation time of each insert, relative to the time of query, increases by \(i\) for each insert. In other words, though the total number of inserts in the stream is \(n\), the difference between the time of query initiation and insert initiation will grow larger for each consecutive insert. Therefore, in any eventual consistent system that has an upper bound on how long a single point can remain inconsistent, only the inserts in the stream initiated after that time are in danger of being missed; all other data points in the system can be safely assumed to be readable.

This upper bound on how long a single insert can remain inconsistent is denoted by \(\phi\). The range of time between a query initiation and \(\phi\) units of time before the query initiation is referred to as the *uncertain past*; that is, the range of time in which initiated inserts have a chance, however remote, of being missed by the query. The range of time more than \(\phi\) units of time earlier than the initiation of a query is referred to as the *certain past*; that is, the range in which initiated inserts are guaranteed to be committed and readable in time for the query.

Given this system variable \(\phi\) and the insertion rate \(f\) of a data stream, the maximum
number of points in danger of being missed by the query, denoted \( n_0 \), is:

\[
n_0 = \lfloor \phi \cdot f \rfloor
\]  

(6.1)

Therefore, as long as the number of points in the insert stream is \( n \geq n_0 \), increasing \( n \) has no impact on \((t, k)\)-staleness.

The value of \( t \) essentially serves to decrease the range of the uncertain past, since \( t \) ensures that no inserts exist in the range between the time of query initiation and \( t \) units of time prior to the query initiation. With larger values of \( t \), the likelihood of freshness is expected to increase. The same is true for \( f \), since larger gaps between inserts lead to fewer inserts in the uncertain past.

### 6.3.4 Staleness and Error

In key-value PBS, read operations are deemed fresh or stale solely by the number of writes missed by the read operation. In A-PBS, the number of missed insert operations is not the only means of qualifying query staleness. The relative error of an aggregate query can also be used as an indicator of a query’s staleness, which can be useful in understanding the practical impact staleness has on a query’s result and how different the result would be under a perfectly consistent system.

In order to reason about aggregate query error, the notions of a true query answer, and an observed query answer must first be defined:

**Definition 6.3 True aggregate value**

Given a data stream \( \text{DATA}(n, f, D) \), the true aggregate value of a query \( Q \) with aggregation function \( A \) is the result of applying \( A \) to all inserts in the data stream covered by the query’s bounding box.
Definition 6.4 Observed aggregate value
Given a data stream \( \text{DATA}(n, f, D) \), the observed aggregate value of a query \( Q \) with aggregation function \( A \) is the result of applying \( A \) to all inserts in the data stream covered by the query’s bounding box which are currently readable by the query. That is, excluding any insertions which are in transit, and therefore not yet committed to the data store.

One immediate problem that arises when comparing the error of aggregation functions is the variation of the measure values associated with each point. While one system may contain measure values that lie between 0 and 1, another may contain measure values that lie in a much larger range, for example between 1000 and 100000. Because of this variation between data sets, an absolute measure of error which is simply the difference between the true value and the observed value is not an effective measure of result quality. For example, the first system above is likely to have a much smaller absolute error than the second system which has much larger measure values, regardless of consistency, since a missed point in the second system has a much larger impact on the error than even many missed points in the first system.

The definition of aggregate relative error avoids this problem by dividing the absolute error by the true aggregate value, allowing results across different measure distributions to be compared:

Definition 6.5 Aggregate relative error
The aggregate relative error of a query \( Q \) with an observed aggregate value of \( o \) and a true aggregate value of \( v \) is \( \frac{|o - v|}{v} \).

Based on this definition, the notion of \( (t, \varepsilon) \)-staleness is defined which, like \( (t,k) \)-staleness, describes whether a query is acceptably consistent, but with respect to a relative error in the result \( \varepsilon \) rather than a number of insertions \( k \):
Definition 6.6 \((t, \varepsilon)\)-staleness

Given a DATA\((n, f, D)\) stream, a query \(Q\) with aggregation function \(A\), initiated \(t\) units of time after the last insert in the stream, has \((t, \varepsilon)\)-staleness if and only if the aggregate relative error of its result, computed by \(A\), is greater than \(\varepsilon\).

\((t, \varepsilon)\)-staleness essentially places an upper bound \(\varepsilon\) on the relative error of a query. A query whose relative error is less than or equal to \(\varepsilon\) is said to have an acceptable amount of error, in which case the query is acceptably consistent. This is similar to \((t, k)\)-staleness, but where \((t, k)\)-staleness measures staleness depending on whether or not points are present during the time the aggregation takes place, \((t, \varepsilon)\)-staleness measures staleness based on the result of the aggregation. Because of this, \((t, \varepsilon)\)-staleness is dependent on the aggregation function \(A\) used by the query, as well as the distribution \(D\) of measure values in the data stream.

An important difference between \((t, k)\)-staleness and \((t, \varepsilon)\)-staleness is that missed points only impact \((t, \varepsilon)\)-staleness if they have an impact on the final aggregate value. For example, missing a single point will not likely have an impact on \((t, \varepsilon)\)-staleness when \(A = \max\), since that point would have to be the largest point covered by the query in order to impact the result. In contrast, with \(A = \text{count}\), missing a single point is guaranteed to increase the relative error of the query.

6.3.5 Probabilistic Staleness

The definitions presented thus far examine how staleness impacts queries on an individual basis. While this is a useful way to begin to reason about staleness in aggregate systems, the above metrics only describe individual queries within a system, not the system itself. In order to discuss the accuracy or staleness of a system as a whole, \((t, k)\)-staleness and \((t, \varepsilon)\)-staleness can be applied on a probabilistic basis as follows:

Definition 6.7 Bounded \((t, k)\)-staleness

A system for performing aggregate queries on a DATA\((n, f, D)\) stream has bounded \((t, k)\)-staleness with probability \(p\) if and only if, with probability \(p\), an aggregate query \(Q\) with coverage \(C\) does not have \((t, k)\)-staleness.
Definition 6.8 Bounded \((t, \varepsilon)\)-staleness

A system for performing aggregate queries on a DATA\((n, f, D)\) stream has bounded \((t, \varepsilon)\)-staleness with probability \(p\) if and only if, with probability \(p\), an aggregate query \(Q\) with coverage \(C\) and aggregation function \(A\) does not have \((t, \varepsilon)\)-staleness.

These bounded staleness metrics can be used to evaluate whole systems by quantifying their probability of returning unacceptably inconsistent results. As with the corresponding metrics for individual queries, “acceptable” is defined by a given \(k\) or \(\varepsilon\). For example, one might ask what the probability is of a system returning a query result that misses at most \(k = 4\) insertions, or whether a system will, with 99% probability, return query results with at most \(\varepsilon = 0.1\%\) relative error.

6.4 Distributed Aggregate Models

A-PBS is designed to analyze large distributed aggregate query systems, like VOLAP. However, the nature of such systems makes performing experiments at scale expensive, both in terms of time and computational resources. For example, when evaluating various deployment architectures with respect to certain staleness requirements, performing experiments with each potential architecture may not be feasible, since each individual experiment may require loading billions of data elements into a system deployed on a large and expensive set of compute nodes.

A less expensive strategy is required in order to realistically investigate the impact that system parameters have on staleness. Even more so than with PBS, it is difficult to analyze this behaviour with a simple closed form due to the many latency distributions and dependant probabilities involved. Accordingly, this section describes a model which, via a simple simulation, can be used to estimate the probability \(p\) of bounded \((t, k)\)-staleness or bounded \((t, \varepsilon)\)-staleness of a distributed aggregate system given a set of parameters which describe the system.

6.4.1 Unreplicated Aggregate Model

Recall that an OLAP system contains a set of \(d\)-dimensional points with associated measure values, and answers queries which apply an aggregation function to subsets of the contained
The A-PBS model for a distributed OLAP system without replication consists of:

- \( n \) shards which store \( d \)-dimensional point data and their associated measure values.
- \( m \) indices which are used to determine which points are stored in which shards.
- \( c \) clients which send insert and aggregate query operations to an index.

A shard stores data items, and is capable of aggregating all stored measure values within a specified region. An index stores location metadata used to determine which shards store which points, and acts as a router for insert and aggregate query requests. Each shard is responsible for representing a specific subspace of \( d \)-dimensional space, but not uniquely responsible; that is, these subspaces may overlap.

When a client wants to insert into or query the system, it sends a request to an index which in turn sends requests to the relevant shards. Responses from shards are returned to the index, which then responds to the client. If a query covers multiple shards, the index is responsible for aggregating the aggregate results returned by each shard before responding to the client. Accordingly, the aggregate query results from shards are referred to as partial aggregations, and aggregate query results from indices are referred to as complete aggregations. Only complete aggregations are visible to the client.

Note that this model, illustrated in Figure 6.5, omits architectural implementation details such as which compute nodes are responsible for certain tasks. For example, in VOLAP, a server serves as a single index, and a worker is responsible for hosting several shards. However, this model does not contain any concept of servers, workers, or other compute nodes, and is applicable to systems that use a variety of architectures.
6.4.2 Quorum-replicated Aggregate Model

In order to apply the idea of quorum-based replication to distributed aggregate systems, the simple model is extended with redundant or replicated shards. Since shards must now be grouped by the subset of data they replicate, this extended model has \( n \) shard sets, rather than \( n \) individual shards, where a shard set is a group of shards which all replicate the exact same subspace of \( d \)-dimensional space. Each shard set is composed of \( N_r \) replicas, so the model has \( n N_r \) shards in total. Shard set member \( D_s^i \) is the \( i \)th replica of the shard \( s \). Figure 6.6 illustrates this extended model for an aggregate quorum-based system. Note that the dashed boxes around shard sets do not imply that shards are stored on single nodes, but indicate a set of structures which replicate the same data. Shard replicas would typically be distributed among nodes to provide redundant storage, load balancing, and fault tolerance.

With quorum-based replication, the routing of inserts and queries by indices becomes more complex. Instead of an index routing an insert to a single shard, each insert must now be routed to all \( n \) shards replicating the point contained by the query. Rather than reporting to the client that the insert has completed after a single shard reply, the index waits for \( W \) replies.

Quorum-based queries are somewhat more complicated. For each shard set relevant to the query’s box, the query is sent to \( N_r \) shard replicas. Once the index receives at least \( R \) partial aggregations from each relevant shard set, it selects the “best”2 partial aggregation.

---

2. The precise meaning of “best” in this context is explained in Section 6.4.3.
Figure 6.6: Extended model for distributed OLAP with quorum-based replication. Clients $C_1$ to $C_c$ each communicate with one of $m$ indices, which communicate with as many as $n$ groups of shards, each group containing $N_r$ replicas. Dashed boxes correspond to groups of structures which replicate the same structure.

from each shard set. After selecting the partial aggregations to use for each shard, the index aggregates these partial aggregations and sends the complete aggregation to the client. Figure 6.7 illustrates the flow of messages for a single query in this model.
6.4.3 Combination of Partial Aggregations

In a system with aggregate quorum consensus, waiting for $W$ replies after a write is used as a way to force the client to wait until the write has been written to $W$ replicas before considering the insert sufficiently readable. $W$’s impact on consistency is straightforward: increasing $W$ results in greater insert latencies but also in greater confidence that the insert has been written to more replicas in the system.

Waiting for $R$ read responses is more complex. By waiting for $R$ read responses, there are $R$ different partial aggregations per bucket set with likely varying degrees of consistency. This introduces the problem of deciding which partial aggregations are the most correct and should be used in the construction of the complete aggregation.

To facilitate analysis of systems with aggregate quorum consensus, this section describes a set of rules for combining partial aggregations using only the value of the partial aggregation itself, for each of the following six aggregation functions commonly used in OLAP systems: count, sum, max, min, mean and stdev [40, 57].

The count aggregation function is the simplest case. Given the assumption that points may only be inserted to the system, that is, the lack of a delete operator as in a typical OLAP system, partial aggregate selection becomes trivial. Each relevant insert adds a value of 1 to the result of the aggregation regardless of measure value, and points are never removed from the system, so the true aggregate value of a count query never decreases over time. An aggregation function that follows this property is monotonic:
**Definition 6.9 Monotonically increasing aggregation function**

An aggregation function is monotonically increasing if and only if the result of the aggregation never decreases as more points are added.

**Definition 6.10 Monotonically decreasing aggregation function**

An aggregation function is said to be monotonically decreasing if and only if the result of the aggregation never increases as more points are added.

If an aggregation function is monotonic, then combining $R$ different partial aggregations is trivial. Since incorrect queries in this model can only arise from missing points, for monotonically increasing aggregation functions, a partial aggregate result is guaranteed to be less than or equal to the true partial aggregate value. The partial results received thus serve as lower bounds on the true partial aggregate value, and therefore the result with the greatest value is the most correct. A similar argument applies for monotonically decreasing aggregation functions, where instead the result with the smallest value is the most correct.

The $\min$ and $\max$ aggregation functions are, like $\text{count}$, monotonic, and therefore it is straightforward to select the most correct of multiple partial aggregations. Since $\min$ is monotonically decreasing, the smallest partial aggregate value is the most correct. Similarly, since $\max$ is monotonically increasing, the largest partial aggregate value is the most correct.

The $\text{sum}$ aggregation function is more complex, since its behaviour depends on the distribution $D$ from which the measure values are drawn. If $D$ is known to yield only values $\geq 0$, then $\text{sum}$ is monotonically increasing. Similarly, if $D$ is known to yield only values $\leq 0$, then $\text{sum}$ is monotonically decreasing. However, if $D$ yields both negative and positive values, then $\text{sum}$ is not monotonic, and selecting the most correct partial aggregate value is more difficult. The experiments here use the following heuristic, which depends on the mean of $D$: If the mean is $> 0$, then given an infinite insert stream, the value of $\text{sum}$ will approach $\infty$, so select the largest partial aggregate value. Similarly, if the mean is $< 0$, select the smallest partial aggregate value. If the mean is 0, then the value of $\text{sum}$ will approach 0 as more points are added to the system, so select the partial aggregation closest to 0. Since only the sign of the mean is relevant to the heuristic, and the mean is simply $\text{sum}$ over $\text{count}$, the sign of the mean can be estimated by taking the sign of the $\text{sum}$ partial aggregation.
The mean and std dev aggregation functions are the most difficult, since neither are monotonic under most circumstances. In this case, another heuristic is used based on the value of the count aggregation in the same space: the partial aggregation with the largest count value is the one with the most samples, and is therefore the most likely to be closest to the true mean or standard deviation.

Note that although the maximum count heuristic would likely select the most correct partial aggregation for many aggregation functions, it does not guarantee correctness for functions like min or max where a correct selection is possible. For example, a partial aggregation $A$ could include the maximum element but have fewer elements than another partial aggregation $B$ which does not, so in this case $A$ should be chosen despite $B$ having a higher count.

### 6.5 Simulation

In order to understand how the different parameters affect staleness within these models, and by extension real OLAP systems, a method for evaluating the probability $p$ of bounded $(t,k)$-staleness and bounded $(t,\varepsilon)$-staleness is needed. To accomplish this, a Monte Carlo simulation is performed to simulate several trials consisting of an insert stream followed by a query in the distributed aggregate model described in Section 6.4.2. The result of each trial is whether or not the query has $(t,k)$-staleness or $(t,\varepsilon)$-staleness. Then, given the set of trial results, probabilities for bounded staleness can be determined.

To minimize computation, any insert stream size $n > n_0$ is simulated as a stream of size $n_0$, the maximum number of points in danger of being missed by a query given by Equation 6.1. This is possible because setting $n > n_0$ does not affect bounded $(t,k)$-staleness by definition, and only positively affects bounded $(t,\varepsilon)$-staleness since larger $n$ causes relative errors to decrease.

#### 6.5.1 Simulation Parameters

The time at which each simulated insert in the stream is consistent at each shard replica is determined by sampling from a set of simulation parameters which describe the performance characteristics of the system. The same set of parameters is sampled to determine the time a query arrives at each shard in the system, and the set of $R$ shards from each shard set which
are the first to arrive at an index. Comparing the insert commit times against the query arrival times can then be used to determine the number of stale inserts. A summary of the simulation parameters is given in Table 6.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(I \rightarrow S)$</td>
<td>Distribution of time taken to send a message from an index to a shard</td>
</tr>
<tr>
<td>$L(S \rightarrow I)$</td>
<td>Distribution of time taken to send a message from a shard to an index</td>
</tr>
<tr>
<td>$T_w(I)$</td>
<td>Distribution of time taken for local insert work on an index</td>
</tr>
<tr>
<td>$T_w(S)$</td>
<td>Distribution of time taken for local insert work on a shard</td>
</tr>
<tr>
<td>$T_r(I)$</td>
<td>Distribution of time taken for local query work on an index</td>
</tr>
<tr>
<td>$T_r(S)$</td>
<td>Distribution of time taken for local query work on a shard</td>
</tr>
</tbody>
</table>

Table 6.1: Simulation parameters

$L(I \rightarrow S)$ is a distribution which describes the latency of sending a message from an index to a shard. Likewise, $L(S \rightarrow I)$ is a distribution which describes the latency of sending a message from a shard to an index. This model assumes that network latency does not change significantly for different types of messages, and therefore network latencies for any type of message (insert, query, or reply) sent within the system can be drawn from $L(I \rightarrow S)$ or $L(S \rightarrow I)$.

$T_w(I)$ and $T_w(S)$ are distributions which describe the time taken for an index (I) or shard node (S) to complete the local computation required for an insert (write). For indices, this is the time taken to parse the incoming insert message, plus the time taken to update its location metadata, plus the time taken to determine the location of the shard replicas that receive the insert. For shards, this is the time taken to parse the incoming insert message, commit the insert, and construct a reply that will be sent to the index.

Similar to $T_w(I)$ and $T_w(S)$, $T_r(I)$ and $T_r(S)$ are distributions which describe the time taken for an index or shard to complete the local computations required for a query (read). For indices, this includes the time taken to retrieve the list of all shards which must be queried and the time taken to construct the message for each shard. For shards, this includes the time taken to perform the aggregation for the query and to construct a reply message.

Given these simulation parameters, the maximum amount of time for an insert to remain stale is $\phi = \max(T_w(I)) + \max(L(I \rightarrow S)) + \max(T_w(S))$. 
6.5.2 Algorithm

Each trial in the simulation begins by calculating the number of inserts in the uncertain past of the stream, which is trivial given Equation 6.1. Then, for each insert in the uncertain past:

1. The start time is calculated as $-(ci + t)$, where $c$ is the number of future inserts in the stream, and $i$ is $1/f$, the time between inserts in the stream.

2. The shard set the insert belongs to is determined by randomly sampling from a uniform integer distribution between 0 and $n - 1$.

3. The commit time of the insert on each relevant shard is simulated by determining the duration of time until committal, which is obtained by sampling $L(I \rightarrow S)$, $T_w(I)$, and $T_w(S)$, then adding the sum to the start time.

4. The quorum reply time (the time at which the index has received the write reply from the shard) is computed by adding a sample from $L(S \rightarrow I)$ to the shard replica commit time.

The simulation process for an insert is illustrated in Figure 6.8.

---

Figure 6.8: Simulation of a single insert with $N_r = 3$ and $W = 2$. Key times recorded for later use during query simulation are marked with asterisks.
Once the insert stream of a trial has been simulated, the results are stored and used to conduct the query simulation:

1. First, the earliest time that the query can be initiated while still observing write quorum rules and the value of \( t \) is determined. To do this, the time of the \( W \)th earliest quorum reply time for each insert is retrieved, and the maximum of these is recorded as the earliest time all write quorums have met their minimum requirement of receiving \( W \) replies for each insert. Adding \( t \) to this value yields the earliest query initiation time.

2. A random sample of \( T_r(I) \) is added to the earliest query initiation time to determine the earliest time at which the index has completed local work, such as looking up shard locations.

3. For each shard, the time of query arrival is determined by adding a sample of \( L(I \rightarrow S) \) to the time the index has finished its local work. While this value is useful for determining freshness since the commit time for each insert for each shard is also known, further simulation is required to determine which shards are the first \( R \) responders.

4. For each shard, the time of query arrival is offset by a random sample from \( T_r(S) \) and \( L(S \rightarrow I) \) to determine the time that the index receives the response from that shard. From this, the time \( R \) responses have been received from each shard set is computed, and the maximum is taken to get the time the query has met its read quorum requirements for each shard set.

The simulation process for a query is illustrated in Figure 6.9.

Given these timing results, shards are grouped into two sets: those which responded to the query before \( R \) responses were received from each shard set, and those which have not. Since those that have not responded in time are not included in the result of the query, they may safely be ignored.

For all shards which responded in time, an aggregation is simulated for all inserts within the query’s coverage, with the exception of inserts that were committed locally after the query’s arrival. For \((t, \varepsilon)\)-staleness, the provided aggregation function is used, and for \((t,k)\)-staleness, the \texttt{count} function is used. The partial aggregation of each shard is combined using the rules described in Section 6.4.3, then all combined partial aggregations
are aggregated into a single observed aggregate value. This aggregation process is then repeated without excluding any inserts to compute the true aggregate value, the expected result with a perfectly consistent system.

Given these values, whether or not the query has $(t, \varepsilon)$-staleness can be determined by calculating the error between the true and observed value. Similarly, $(t, k)$-staleness can be determined by comparing the difference between the observed and true counts with $k$.

### 6.6 Experimental Evaluation

#### 6.6.1 Simulation Validation

To validate the accuracy of the A-PBS model and simulation, the actual probability of bounded $(t, k)$-staleness observed for VOLAP is compared with the probability of bounded $(t, k)$-staleness predicted by the simulation. The simulation parameters $T_v(I)$, $T_w(S)$, $T_r(I)$ and $T_r(S)$, which describe the time taken to execute an insert or query operation on an index
or shard, are obtained by measuring the time taken to perform these operations in VOLAP.

These latency distributions were measured with VOLAP running on 8 Amazon EC2 c4.xlarge nodes. Write latencies were recorded while ingesting 100000 elements from the TPC-DS data set with 8 hierarchical dimensions. Read latencies and the observed probability of bounded \((t,k)\)-staleness were recorded by issuing 50\% coverage count queries \(t\) units of time after inserting 100000 elements. The first 99 percentiles of these distributions are shown as cumulative distribution functions in Figure 6.10.

![Cumulative distribution functions](image)

Figure 6.10: Cumulative distribution functions for distributions \(T_w(I)\), \(T_w(S)\), \(T_r(I)\) and \(T_r(S)\) observed in VOLAP. For legibility, only the first 99 percentiles are shown.

To determine the actual probability of bounded \((t,k)\)-staleness, the system is first sent a stream of 100000 insertions. Once \(t\) units of time elapse after \(W\) quorum responses are received, a count query covering the entire database is sent. The difference between the number of inserts sent and the result of the query is the number of points missed by the query, which determines whether the query has \((t,k)\)-staleness. This process is repeated 100 times to generate an approximate probability of bounded \((t,k)\)-staleness.

Figure 6.11a plots the simulated and observed VOLAP probabilities of bounded \((t,k=0)\)-staleness, with quorum configuration \((N_r=1, W=0, R=1)\). This configuration reflects the throughput-maximizing configuration without strong serialization where operations are executed in parallel as soon as possible without waiting for write quorum consensus, as investigated in Section 5.8. The x-axis represents \(t\), the amount of time elapsed in milliseconds between the final insert transmission and the query. The simulation closely models the behaviour of VOLAP for each value of \(t\), and accurately predicts a near 100\% probability of bounded \((t,k=0)\)-staleness after \(t = 15\) milliseconds. The simulated and observed number of missed points under the same quorum configuration are shown in Figure 6.11b.

Similarly, Figure 6.12 shows the simulated and observed probabilities of bounded
Figure 6.11: A-PBS simulation compared to VOLAP for ($N_r=1, W=0, R=1$)

$(t,k=0)$-staleness and the average number of missed inserts, but with quorum configuration ($N_r=3, W=0, R=1$), that is, a replication factor of $N_r = 3$. Since $W = 0$, one might not expect the consistency to change compared to the previous experiments. However, even with $W = 0$, increasing $N_r$ reduces insertion throughput, which results in a corresponding increase in the observed probability of bounded $(t,k)$-staleness. When the ingestion rate parameter of the simulation is adjusted to match that of VOLAP, as it is here, the simulation accurately predicts the behaviour of the system.

This shows that the simulation can be used to make accurate predictions about system behaviour, though care must be taken to ensure that the simulation parameters reflect the performance of the system under various configurations. In the case of VOLAP, for example, the ingestion throughput must be reduced slightly as the replication $N_r$ increases to obtain highly accurate results. Such inter-dependencies are deliberately not included in the model itself, since they are inherently system-dependent, and A-PBS seeks to be a general framework for evaluating aggregate query systems. However, it is straightforward to determine these relationships for a given system via preliminary experimentation and choose the simulation parameters accordingly.
Figure 6.12: A-PBS simulation compared to VOLAP for \((N_r=3, W=0, R=1)\)

### 6.6.2 Simulation Parameters

The remainder of this section investigates the impact that different quorum configurations and system parameters have on bounded \((t, k)-\)staleness and bounded \((t, \varepsilon)-\)staleness through simulation.

Except where otherwise specified, the simulation and system parameters are as follows: The latency distributions are those sampled from VOLAP as described in the previous section. The simulated system has \(m = 128\) bucket sets distributed across 32 bucket nodes. The insertion rate is 50000 insertions per second with \(N_r = 1\) replicas, with a decrease of 2500 for every additional replica, which is a conservative simplification of the behaviour observed in VOLAP. Measured queries have 100% coverage for strict measurement of potential staleness and error. The data stream contains \(n_0 = 1192\) inserts, calculated from the insertion rate and \(\phi \approx 24\) ms derived from the measured latency distributions as described in Section 6.5.1.

### 6.6.3 Impact of Write Quorums

The results above show that even a very relaxed quorum configuration like \((N_r=1, W=0, R=1)\) can provide acceptable freshness bounds for many applications which prioritize performance over freshness. However, configurations with \(W = 0\) may not meet the needs of applications with stricter requirements. The impact that quorums have on freshness and error is demonstrated in Figure 6.13, which shows bounded \((t, k)-\)staleness and the average
number of missed points for various quorum configurations.

![Graphs showing bounded \((t,k=0)\)-staleness and expected number of missed points](image)

Figure 6.13: The impact of quorums on freshness and error

The \((N_r=1, W=0, R=1)\) configuration has effectively zero probability of bounded \((t,k)\)-staleness in the first few ms since when \(W = 0\), the query is issued immediately after the last insertion without waiting for a response. Due to the high rate of insertion, the number of missed points seen in Figure 6.13b is initially nearly 100, and gradually decreases until bounded \((t,k)\)-staleness is achieved and no more points are missed approximately 14 ms after the last insert was issued.

Note that these measurements are performed at the client, so this potential wait directly impacts the observed freshness. This accounts for the large disparity between \(W = 0\) and the other results, since when \(W > 0\), the client must wait for a response from the system before sending another request. For example, since the system is highly parallel, the difference in time between waiting for 2 responses rather than 1 is much less than the difference between waiting for any response rather than none at all.

In contrast, all configurations with \(W > 0\) achieve bounded \((t,k)\)-staleness more rapidly relative to \(t = 0\), which corresponds to the time when write quorum is achieved, that is, the earliest time the system allows a query to be issued. The configurations with \(W > 0\) miss so few points that visual comparison is difficult because the plot is scaled to such a large range to cover the \(W = 0\) case. Accordingly, the remainder of this section omits \(W = 0\) configurations, and focuses on the impact of quorum configurations suitable for applications with stricter requirements for bounded \((t,k)\)-staleness or bounded \((t, \varepsilon)\)-staleness.
6.6.4 Bounded Staleness

Figure 6.14a shows the probability of bounded \((t,k=0)\)-staleness as a function of \(t\) for several quorum configurations. Each plotted point corresponds to a single simulation composed of 1000 trials.

![Graphs showing probability of bounded \((t,k)\)-staleness for various quorum configurations.](image)

Figure 6.14: Probability of bounded \((t,k)\)-staleness for various quorum configurations

Quorum configuration \((N_r=3, W=1, R=1)\) yields the lowest probability of bounded \((t,k)\)-staleness, because it has the largest number of replicas and the smallest read and write quorums of the set. The probability of consistency increases as the number of replicas is reduced. For example, with \((N_r=2, W=1, R=1)\), each insert is guaranteed to be written locally on at least 50% of replicas instead of 33%, increasing the probability that a query achieves consistency.

With the latency distributions of this system, configurations with \(W < 3\) generally
have a much higher probability of consistency. All configurations with \( N_r = 2 \) offer better consistency than those with \( N_r = 3 \) regardless of quorum configuration, though this is system-depandanct and not generally true. For example, Figure 6.15 shows the same measurements for a hypothetical system with balanced read and write latencies which follow an exponential distribution with \( \lambda = 0.1 \). On such a system, better consistency is achieved by both \( (N_r=3, W=1, R=2) \) and \( (N_r=3, W=2, R=1) \) compared to \( (N_r=2, W=1, R=1) \).

![Figure 6.15: Probability of bounded \((t,k=0)\)-staleness with balanced read/write latencies](image)

The quorum configurations \( (N_r=1, W=1, R=1) \) and \( (N_r=2, W=2, R=1) \) are perfectly consistent, since \( W = N_r \) write replies ensures that each insert is written to all replicas before the query begins. Configuration \( (N_r=2, W=1, R=2) \) comes very close, but is not perfectly consistent, though this is difficult to see visually for the VOLAP model, but more clear in Figure 6.15. In key-value quorum systems, \( (N_r=2, W=1, R=2) \) constitutes a complete, intersecting quorum, and should therefore always be consistent. However, this property does not hold in aggregate systems. Though \( (N_r=2, W=1, R=2) \) ensures that all writes are included in the set of partial aggregations received on the index, the combination of partial aggregations may force the selection of incorrect partial aggregations.

Plots for the slightly more lenient cases of bounded \((t,k)\)-staleness where \( k = 1, 2, \) and 3 are presented in Figures 6.14b, 6.14c, and 6.14d, respectively. When \( k = 1 \), the difficult configurations with \( N_r = 3 \) achieve consistency approximately 2 ms earlier. When \( k \geq 2 \), \( (N_r=3, W=1, R=2) \) achieves near-perfect consistency, though \( (N_r=3, W=1, R=1) \) sees only a marginal improvement.

Figure 6.16 shows the probability of bounded \((t,k=0)\)-staleness for varying coverages. As one would expect, queries with the highest coverage see the worst probability of bounded
(t, k)-staleness, but eventually all queries achieve consistency.

Figure 6.16: Probability of bounded (t, k=0)-staleness for various coverages

Figure 6.17 illustrates the number of points missed by a query. Note that the worst quorum configuration, (N_r=3, W=1, R=1), misses only one point on average at t = 0 ms, despite having a relatively poor probability of bounded (t, k=0)-staleness. Since queries in aggregate systems typically aggregate many thousands of points, it is relatively unlikely that the result of a query which does not achieve (t, k=0)-staleness differs significantly from that of a query which does. This highlights the need for an error-based metric which can better show the actual impact that inconsistency has on query results.

Figure 6.17: Number of missed points for various quorum configurations
6.6.5 Latency

Figure 6.18a shows the average latency of inserts with various quorum configurations. Here, comparing quorum configurations \((N_r=1, W=1, R=1)\), \((N_r=2, W=1, R=1)\), and \((N_r=3, W=1, R=1)\) shows that insert latency decreases as the replication factor \(N_r\) increases. This is because an insert’s latency is determined by the insert latency of the fastest \(W\) of \(N_r\) replicas, and increasing \(N_r\) therefore allows for more samples from the insert latency distribution. This reflects well-designed real systems where replicas are evenly distributed among compute resources, allowing for improved performance. Increasing the value of \(W\) has a negative impact on latency since the system must wait for more replicas before an insert is complete.

Similarly, Figure 6.18b shows the average latency of queries with various quorum configurations. The effects on query latency for different quorum configurations is similar to that of insert latency: greater values of \(N_r\) decrease latency, as the query waits for only the first \(R\) replies from each bucket set. Likewise, greater values of \(R\) increase latency.

Note that Figure 6.18a and Figure 6.18b have different vertical scales, with the average read latency being roughly 4 times the average write latency. This is because, in this simulation, queries have 100% coverage and so must wait for a reply from at least \(nR\) different replicas, where \(n = 128\) bucket sets.

Figure 6.18: Average latency for various quorum configurations
6.6.6 Bounded Error

Unlike bounded staleness, evaluation of bounded error depends on the distribution of the measures in the database. To emulate different types of data, here a folded normal distribution with mean \( \mu = 0 \) and standard deviation \( \sigma = 1 \) is used to represent a short-tailed distribution of measure values, and an exponential distribution with \( \lambda = 1 \) to emulate fat-failed distributions.

Based on the results in the previous section, this section investigates a quorum configuration of \((N_r=3, W=1, R=1)\) except where otherwise noted, since it has the weakest consistency guarantees. Three representative aggregation functions are explored: \( \text{sum} \), since it is monotonically increasing with a steady rate of change regardless of the number of points (similar to \( \text{count} \)), \( \text{max} \), since its value is essentially determined by a single point (similar to \( \text{min} \)), and \( \text{mean} \), since it is non-monotonic and its change drops as the number of points in the aggregation increases (similar to \( \text{stddev} \)).

Figure 6.19 shows the probability of bounded \((t, \varepsilon=0)\)-staleness for the aforementioned distributions and aggregation functions. This is somewhat similar to the probability of bounded \((t, k=0)\)-staleness from Figure 6.14a, in that there is no tolerance for error. However, where bounded \((t, k)\)-staleness views error as missing any points within the query, bounded \((t, \varepsilon)\)-staleness views error in terms of the relative error of the observed and true aggregate results. Because of this, the \( \text{max} \) aggregation function easily achieves near 100% probability of bounded \((t, \varepsilon=0)\)-staleness, despite the lower probability of bounded \((t, k=0)\)-staleness of the system as a whole. This is because missing a point in a \( \text{max} \) query only impacts the final aggregation if that point’s measure value is the largest in the query’s coverage. Conversely, missing a single point with a positive value as a measure for the \( \text{sum} \) aggregation function, or any point at all for the \( \text{mean} \) aggregation functions will result in a non-zero relative error. This is why the curve for \( \text{sum} \) and \( \text{mean} \) precisely matches the corresponding \((N_r=3, W=1, R=1)\) curve for bounded \((t, k=0)\)-staleness in Figure 6.14a.

In Figure 6.19b, \( \varepsilon \) is increased to 0.001, that is, a small amount of slack is given for relative error. This makes the impact that measure distribution and aggregation function have on bounded \((t, \varepsilon)\)-staleness clear. As expected, \( \text{max} \) retains its near 100% probability of bounded \((t, \varepsilon)\)-staleness. The probability for \( \text{mean} \) and \( \text{sum} \), however, have separated from each other, with \( \text{mean} \) yielding higher probabilities than \( \text{sum} \). This is because the \( \text{mean} \) function is less sensitive to missing values, especially as the number of points in the data
stream increases. For sum, the choice of measure distribution has little impact on bounded \((t, \varepsilon)\)-staleness, while the exponential distribution yields slightly lower probabilities of bounded staleness than the folded normal distribution when the mean aggregation function is used.

![Figure 6.19: Probability of bounded \((t, \varepsilon)\)-staleness with various aggregation functions and measure distributions for \((N_r=3, W=1, R=1)\)](image)

The impact of quorum configuration on bounded \((t, \varepsilon)\)-staleness can be seen by comparing Figure 6.19 with Figure 6.20, which shows corresponding plots for \((N_r=3, W=1, R=2)\), that is, with an increased read quorum \(R=2\). Here, the probability of bounded \((t, \varepsilon)\)-staleness is significantly higher for all aggregation functions and measure distributions, as expected given the staleness figures in Section 6.6.4. However, this configuration would have significantly increased latency and reduced throughput due to the increased time the system must wait for read quorum replies, demonstrating the fundamental trade-off between consistency and performance.

The contrast between these results highlights the utility of being able to quickly estimate the probability of bounded \((t, \varepsilon)\)-staleness of a given quorum configuration. For example, this system with \((N_r=3, W=1, R=1)\) would not suit the needs of a very high-frequency application which requires at least 90% probability of bounded \((t, \varepsilon)\)-staleness for mean aggregation after 1 ms, but would be suitable with \((N_r=3, W=1, R=2)\), justifying the performance cost of the increased \(R\). In contrast, an application which only requires 80% probability of bounded \((t, \varepsilon)\)-staleness for mean aggregation after 5 ms would be easily served by the \((N_r=3, W=1, R=1)\) configuration, and could benefit from the increased
performance when $R = 1$.

Figure 6.20: Probability of bounded $(t, \varepsilon)$-staleness with various aggregation functions and measure distributions for $(N_r=3, W=1, R=2)$

The average relative error for each distribution and aggregation function with $(N_r=3, W=1, R=1)$ is shown in Figure 6.21. As expected, the sum aggregation shows the highest relative error with both distributions, followed by mean, then max, reflecting the results shown in Figure 6.19b.

Figure 6.21: Expected relative error for $(N_r=3, W=1, R=1)$

### 6.6.7 Impact of Insert and Query Latencies

The bulk of this section has used the A-PBS model to investigate how various quorum configurations affect consistency for a particular modelled system, but simulation is also a
useful tool to investigate how performance characteristics of a system in general can impact consistency.

For example, Figure 6.22 shows the impact that a system’s read and write speeds have on consistency by plotting the probability of bounded \((t, k=0)\)-staleness with different query \((T_r(I), T_r(B))\) and insert \((T_w(I), T_w(B))\) distributions generated to model systems with significantly different performance characteristics:

- **Fast reads** uses an exponential distribution with \(\lambda = 0.1\) (mean of 10 ms) for the query distributions, and an exponential distribution with \(\lambda = 0.04\) (mean of 25 ms) for the insert distributions, making queries \(\approx 2.5\) times faster than inserts.
- **Fast writes** uses \(\lambda = 0.1\) for the write distributions and \(\lambda = 0.04\) for the read distributions, making inserts \(\approx 2.5\) times faster than queries.
- **Fast reads and writes** uses \(\lambda = 0.1\) for both the read and write distributions, so both insertions and queries have approximately equal latencies.

![Graph showing probability of bounded (t,k=0)-staleness with varying read/write speeds and (N_r=3, W=1, R=1)](image)

Figure 6.22: Probability of bounded \((t, k=0)\)-staleness with varying read/write speeds and \((N_r=3, W=1, R=1)\)

The impact that the relationship between query and insert latency has on bounded \((t,k)\)-staleness is apparent. When inserts are significantly faster than queries, more inserts are likely to become accessible by a query in the extended amount of time the query takes to be processed at the index, leading to high probabilities of bounded \((t,k)\)-staleness. With faster queries, the opposite is true: queries spend less time being processed at the index and thus arrive at buckets to aggregate insertions earlier, leading to lower probabilities of bounded \((t,k)\)-staleness.
6.7 Summary

Aggregate Probabilistically Bounded Staleness (A-PBS) is a measure for the staleness of aggregate queries. Inspired by the Probabilistically Bounded Staleness (PBS) measure [5] for key-value stores, A-PBS measures staleness for aggregate queries in distributed OLAP systems that may aggregate a large set of data items and depend on the write/read history of the different copies of all data items in a $d$-dimensional bounding box.

A-PBS includes a formal model for describing an OLAP system’s data stream and the state of consistency for individual aggregate queries. A-PBS uses both the number of missed inserts and the relative numerical error of the query result to quantify staleness. A generic model and corresponding Monte Carlo simulation of data aggregation in quorum-replicated distributed OLAP systems complements A-PBS. Given a list of system parameters, this model and simulation can be used to estimate staleness for aggregate queries, thereby enabling the exploration of the trade-offs between consistency and latency in quorum-replicated distributed OLAP systems.

The accuracy of A-PBS has been confirmed by a comparative evaluation with VOLAP, where the staleness of aggregate queries predicted through A-PBS and Monte Carlo simulation was very close to the actually observed staleness of aggregate queries in VOLAP. For example, the difference between the simulated and observed average number of missed data items in an aggregate query covering 100,000 data items was 0.5.

A-PBS analysis also confirms previous observations that relatively relaxed quorum configurations are “good enough” in practice for most applications. For example, with $(N_r=3, W=0, R=1)$, even very large aggregate queries that cover the entire database and are issued immediately after the last insert have $\approx 40\%$ probability of zero staleness. If staleness occurs for such aggregate queries, the expected number of missed data items is only 0.8, resulting in only a very small numerical error in the aggregate query result for sum aggregation, and very close to zero probability of any numerical error for max aggregation.
Chapter 7

Summary and Conclusions

An elastic distributed system designed around an appropriate spatial index can provide good performance for diverse real-time OLAP workloads, without requiring the generation of a data warehouse or snapshot for analysis. The overhead inherent in previous approaches to this problem can be overcome to create a system which can more realistically meet performance requirements. In particular, the DC-tree’s benefits of hierarchy awareness and support for many dimensions can be realized without a considerable reduction in ingestion throughput compared to less capable structures.

Such a distributed system can provide serializable execution for an operation stream, the strongest possible guarantee for query result consistency, with only a modest impact on performance. When serializable execution is unavailable or disabled for performance reasons, the system can provide suitably fresh results for most real-time analytical applications. In this context, the accuracy of results can be predicted via simulation using a general model of quorum-based aggregate data stores. Such a model enables the cost-effective investigation of the impact that system parameters have on result freshness and performance, since these trade-offs can be evaluated without the expense of performing many experiments on a real system.

Experiments show that the approach described here is realistic, and can provide good performance for important workloads. In particular, applications which rely on performing very large aggregations of a database which changes frequently could benefit from these methods.

7.1 Future Work

7.1.1 Improved Spatial Region Representations

The improved performance of the Hilbert PDC-tree is in large part due to the improved MDS representation which eliminates the need for auxiliary data structures during insertion.
However, further improvements could be made in this area.

A limitation of the MDS approach is poor support for large contiguous ranges, since, unlike an MBR, an MDS is a set of values. For example, with a date dimension where year is the coarsest level, a query that aggregates values for every year from 1900 through 2000 would contain a hundred year IDs which impacts performance due to the increased cost of comparison. A hybrid approach which compresses contiguous ranges could mitigate this issue while still providing the overlap-minimizing benefits of MDSs. Similarly, while this work is specifically targeted at hierarchical data, some data sets naturally use some continuous dimensions. An improved representation that supports both hierarchical and continuous dimensions would allow the Hilbert PDC-tree and VOLAP to perform well with a wider range of data sets.

The other main drawback of MDSs compared to MBRs is space consumption. This has the obvious drawback of high memory consumption, but can also result in reduced performance since comparison of large MDSs is more expensive. However, there is opportunity to improve this situation by taking advantage of the tree structure. For example, if a directory node contains only the value 1.2.3 in dimension 1, then it is known that all MDSs in the subtree rooted at that node have values that begin with 1.2.3 in dimension 1. Such an implementation would be considerably more complex, since MDSs would require context and no longer be self-contained, but could significantly reduce the space overhead of the Hilbert PDC-tree.

### 7.1.2 Hierarchy-Aware Space-Filling Curves

The Hilbert PDC-tree takes the approach of manipulating values before the Hilbert mapping to account for hierarchical issues. However, the Hilbert mapping itself is conventional. This approach works well, but a more sophisticated algorithm which takes the hierarchical nature of data into account in the mapping itself may be able to improve performance. Since the structure of the tree is entirely dictated by this mapping, a revised insertion algorithm in conjunction with a suitable mapping may be able to provide stronger theoretical guarantees about query performance, which have traditionally eluded this type of spatial index data structure.
7.1.3 Improved Tree Performance and Splitting Strategies

The most significant source of thread contention in the Hilbert PDC-tree is node splitting. The splitting algorithm must be carefully designed to ensure correctness, and handling the possibility of many threads attempting to split nodes along the same path results in significant locking and copying overhead. These issues could be avoided by pessimistically splitting full nodes during the downward search, thus eliminating the upward split phase entirely. This approach is not typically taken because it produces a less space-efficient tree, but in the context of a multi-threaded structure designed to prioritize high ingestion throughput like the Hilbert PDC-tree, this trade-off is more reasonable.

Similarly, the Hilbert PDC-tree currently only splits nodes in two, though in some cases splitting into three or even more nodes can be advantageous. The current splitting algorithm evaluates every potential split in a node, so many “good” splits are often known. It would be interesting to investigate heuristics for when splitting a node into many pieces would be beneficial.

7.1.4 Real-Time Load Balancing

The real-time load balancing algorithm used in the VOLAP prototype is relatively simple, aiming only to balance the data load between workers to maximize the amount of data that can be stored in the system. This facilitates the experiments in this work, but is not an ideal approach for many real-world deployments.

However, the architecture is designed to support more sophisticated load balancing algorithms. For example, the system could more aggressively balance the query load between workers, to dynamically compensate for situations where most queries are directed to a subset of workers. Adapting existing load-balancing research to the context of VOLAP would likely improve query performance for a wider range of real-world workloads.

7.1.5 Global Hilbert Ordering

When used for shards, the Hilbert PDC-tree ingests data very rapidly by using the Hilbert curve to distribute data rather than performing geometric calculations. However, the global index used in VOLAP servers does not make use of this technique. Preliminary investigation has shown that using this approach globally is more difficult, since splitting and migration in
a distributed context is dramatically more expensive than in an in-memory tree. Accordingly, VOLAP uses an initial geometric distribution designed to spread space across all workers to account for highly skewed data sets. However, ingestion throughput of the system as a whole could be improved by using the Hilbert curve in servers as well as workers. Further investigation is required to determine whether this is possible without sacrificing the skew tolerance and query performance of the system.

7.1.6 Practical Application

The prototype implementation of VOLAP uses a simplified application model that supports only inserts and aggregate queries. However, real-world databases typically require a more extensive feature set, such as support for standard query languages, snapshots and roll-back, atomic transactions, non-aggregating range queries, and so on. While the model used by VOLAP is useful, the system would be more widely applicable if extended to support these features, further work is required to extend VOLAP or use it as a component of another system to provide a more fully-featured database.
Appendix A

MDS Representation

If a single dimension is stored in an array, there are two reasonable representations for a complete MDS: the flat representation where all IDs in all dimensions are stored in the same array, and the split representation where each dimension is stored in a separate array.

<table>
<thead>
<tr>
<th>1.2</th>
<th>1.5</th>
<th>1.6</th>
<th>1.8</th>
<th>2.4.6</th>
<th>2.9.4</th>
<th>2.10.1</th>
<th>2.10.2</th>
</tr>
</thead>
</table>

Figure A.1: Flat MDS representation

<table>
<thead>
<tr>
<th>1.2</th>
<th>1.5</th>
<th>1.6</th>
<th>1.8</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>2.4.6</th>
<th>2.9.4</th>
<th>2.10.1</th>
<th>2.10.2</th>
</tr>
</thead>
</table>

Figure A.2: Split MDS representation

Each representation has advantages and disadvantages. The flat representation is less fragmented in memory, but requires checking the dimension of each ID during a scan, and insertion is more expensive since more IDs must be shifted. The split representation requires fewer operations during a scan since the dimension is implicit, and insertion is theoretically less expensive, but memory is more fragmented.

Figure A.3 and Figure A.4 show the results of benchmarks for each representation. Each experiment shows the time for an operation as the number of dimensions increases, where each point is the average of many thousands of operations with various randomly generated MDSs. Figure A.3a shows the time to calculate the coverage of one MDS by another, i.e. whether or not one MDS completely contains, partially intersects with, or is disjoint from, another. Figure A.3b shows the time to calculate the amount of volume expansion that would result by combining one MDS with another. For both measurement operations, the split representation is significantly faster.
Figure A.3: Box measurement performance

Figure A.4a shows the time to expand an MDS in-place to include the area covered by another MDS. For this operation, the performance of both representations is very similar. Figure A.4b shows the time to construct a new MDS that represents the union of two other MDSs. Here, the flat MDS is faster, since this operation is a linear scan of the two input MDSs and benefits from the lower memory fragmentation, and the new MDS is constructed by appending and does not need to shift any elements as with the in-place expansion.

Figure A.4: Box manipulation and construction performance

These results suggest that the split representation should perform the best overall, particularly for querying since the coverage operation is faster. However, this is a case where micro-benchmarks are misleading. Figure A.5 shows the performance of a multi-threaded tree using each representation for keys. The tree implementation uses the above-measured
operations during insertion and querying. For both insertions and queries, the flat MDS is the clear winner, showing a significant improvement in throughput. In the context of a multi-threaded data structure, the benefits of the cache-friendly flat layout are more significant. Though the split representation appears faster in micro-benchmarks, its fragmented memory layout harms performance when many MDSs are being used concurrently. These results suggest that an even more fragmented representation, such as a tree-based structure, would not perform well, even though some operations could be theoretically more efficient.

![Figure A.5: Tree performance with various box implementations](image-url)
Appendix B

Synthetic Data Generation

A method for randomly generating data is required in order to test performance with a wide range of data sets. In particular, data skew is an important attribute of a data set which is known to significantly affect the performance of distributed database systems [54]. The ability to quickly generate synthetic data sets with extreme skew thus makes it possible to experimentally verify that a system is able to cope with a diverse range of real-world data sources.

**Definition B.1 Data Skew**

Data skew, or simply skew, is a measure of how unevenly values are distributed within a data set. The precise definition of skew depends on the specific model being used to characterize the data, but in general, high skew indicates that the distribution of values is highly irregular. For example, a completely uniform data set in which every value occurs an equal number of times has no skew, while a data set in which a single value occurs more often than all other values combined has extremely high skew.

Data skew is commonly characterized using Zipf’s Law, a discrete analogue to the Pareto distribution [61] which has been found to accurately describe a wide range of real-world data [2, 77, 76].

**Definition B.2 Zipf Distribution**

Given a population of \( N \) elements and a characterizing skew parameter \( s \), the frequency of elements of rank \( k \) is:

\[
f(k; s, N) = \frac{k^{-s}}{\sum_{n=1}^{N} \frac{1}{n^s}}
\]

Generating a flat data set that follows a Zipf distribution is straightforward, however, the hierarchical nature of OLAP data introduces difficulties which must be accounted for in order
to produce reasonable values. The following section describes how random hierarchical values can be efficiently generated, where both the entire data set, and each level of the hierarchy in isolation, roughly follow a Zipf distribution.

### B.1 Generating Skewed Hierarchical Identifiers

Points within the dimension hierarchies of the data set are referred to using the hierarchical IDs described in Section 3.2. Given a desired data skew $s$, IDs could be naively generated by simply generating an index at each level according to a Zipf distribution with skew $s$. This *recursive skew* approach, however, produces data that does not closely approximate a Zipf distribution since the high skew at lower levels produces gaps where the frequency of elements is too low, producing a distribution like those shown in Figure B.1.

![Figure B.1: Zipf-distributed hierarchical IDs with recursive skew](image)

In order to produce a smoother distribution, the skew for lower levels must be chosen more carefully such that the rightmost value within a given level is approximately as frequent as the leftmost value within its parent’s successor. For example, considering a simple two-level hierarchy where all levels have values less than 100, a smooth distribution requires that 1.99 occur approximately as often as 2.1.

Formally, let $L_l$ be a random variable denoting the selection of index $l$ in level $L$, and, similarly, $M_m$ denote the selection of index $m$ in level $M$ which is an immediate child of $L$.

Suppose $l$ has been chosen, and an index for level $M$ must now be selected. In order to produce a smooth distribution, the probability of the maximum value in $M$ being selected
must be approximately equal to the probability that the next value \( l + 1 \) in its parent level was chosen.

For a general \( l \) and \( m \):

\[
P(M_m \land L_l) = P(L_{l+1})
\]
\[
P(L_l) \cdot P(M_m|L_l) = P(L_{l+1})
\]
\[
P(M_m|L_l) = \frac{P(L_{l+1})}{P(L_l)}
\]

The parent level \( L \) is a Zipf distribution with skew \( s \). A different, lesser skew \( s' \) is required to generate indices at the child level \( M \). This value can be found by solving the above using the definition of the Zipf distribution, but since the Zipf distribution is a discrete analogue of the Pareto distribution, it suffices to use the simpler tail function of the Pareto distribution which greatly simplifies the calculation. The corresponding tail function of the Pareto distribution is \( F(x) = \frac{1}{x^s} \), where \( s \) is the skew parameter as in the Zipf distribution \([61]\).\(^1\)

The desired value is the skew \( s' \) for indices \( m \) in the child level \( M \). Note that the same distribution is used for all values in a given level, and it is given that some index \( l \) has been chosen in the parent level \( L \). That is, the selection of \( m \) is independent of \( l \), so, in this context, \( P(M_m|L_l) = P(M_m) \). Substituting in the above, the skew \( s' \) required for the selection of an arbitrary index \( m \) to have equivalent probability to the selection of \( l + 1 \) can be calculated:

\[
P(M_m) = \frac{P(L_{l+1})}{P(L_l)}
\]
\[
\frac{1}{m'^s} = \frac{1}{(l+1)^s}
\]
\[
m'^s = \frac{(l+1)^s}{l^s}
\]
\[
m'^s = \left( \frac{l+1}{l} \right)^s
\]
\[
s' = \log_m \left( \frac{l+1}{l} \right)^s
\]

Therefore, a smooth distribution can be generated by choosing \( s' \) such that \( \max(m) = |M| \) has equivalent probability of being selected as the following index in its parent level, or

\[
s' = \log_{|M|} \left( \frac{l+1}{l} \right)^s
\]

Generating hierarchical IDs in this way produces a smoother distribution which more closely approximates a non-hierarchical Zipf distribution, as shown in Figure B.2.

---

1. Assuming a minimum value \( x_m = 1 \), and using \( s \) for consistency rather than the conventional \( \alpha \).
Hierarchical IDs with a desired skew can therefore be generated by using one random number generator per level, each with a different skew as calculated above. By using a rejection-inversion based random number generator [44], rather than the table-based methods typically used for small Zipf-distributed data sets, very large hierarchical data sets can be generated in effectively constant space. All experiments on synthetic Zipf-distributed values in this work use data generated in this fashion.

### B.2 Generating Queries

Coverage is the key attribute that affects query performance, since it dictates the fraction of the database that must be considered in order to answer the query. However, other query attributes can significantly affect query performance, including which dimensions are split, the specific set of values selected within a dimension, and whether or not these values are contiguous. A sufficiently large set of queries that varies in all these attributes can be used to evaluate performance with high confidence that the results are applicable to real-world query scenarios, since all performance-critical aspects of queries are tested.

Algorithm B.1 takes a bounding MDS for the database and a query coverage, and returns an MDS with roughly the requested geometric coverage.

Details which account for an insufficient number of available IDs in a dimension are omitted here for brevity. The basic approach of Algorithm B.1 is to choose a random number of dimensions to cut, then, starting with a random dimension, choose a suitable number of IDs to achieve the desired coverage. The desired coverage for an individual dimension
Algorithm B.1: gen_query

Input: \( b \), the bounding MDS for entire database

Input: \( c \), the desired coverage from 0 to 1

Result: An MDS with roughly the desired coverage of the database

1. Let \( n_c = \) a random integer from 0 to \( D - 1 \)  
   \((Number of dimensions to cut)\)
2. Let \( d = \) a random integer from 0 to \( D - 1 \)  
   \((Dimension index)\)
3. Let \( q = \) an empty \( D \)-dimensional MDS  
   \((Result query)\)
4. for \( n_c \) iterations do
5.   Let \( c_d = c \sqrt{n_c} \)  
   \((Target coverage for this dimension)\)
6.   Let \( p = d.0.0.0\ldots \)  
   \((ID prefix)\)
7.   foreach level \( l = 1 \ldots \) the lowest level in dimension \( d \) do
8.     Let \( |l| = \) the number of IDs at level \( l \) in dimension \( d \)
9.     Let \( f = 1.0/|l| \)  
   \((Fraction of this level to cover)\)
10. if \( f \leq c_d \lor \) \( l \) is the lowest level then
11.    Let \( k = c_d / f \)  
   \((Number of IDs to add at this level)\)
12.    Let \( l = k \) unique random indices at level \( l \) in dimension \( d \) within \( b \)
13.    foreach \( i \in I \) do
14.       Insert ID \( p + i \) into result query \( q \)
15. else
16.    \( p := x + \) a random ID at level \( l \)
17.    \( d := (d + 1) \mod D \)
18. return \( q \)

is \( c \sqrt{n_c} \) (where \( c \leq 1 \)) so that the product of the resulting \( n_c \) cut dimensions forms an area with roughly total coverage \( c \). To select discrete IDs for each dimension, the algorithm descends until it reaches a level where IDs are sufficiently fine-grained to not exceed the target coverage. Then, a random set of unique indices at this level is chosen to form the set of IDs in the query MDS for this dimension. If IDs are allocated such that the valid indices in a given level are within a contiguous range, this selection can be performed efficiently using space proportional to the number of selected indices using, for example, Floyd’s Algorithm [9].

Note that the coverage used here is geometric, that is, it describes a fraction of the bounding box, but not necessarily a fraction of the items in the database. The actual coverage of a query may dramatically differ from its geometric coverage. For example, a query with 50% geometric coverage may have near-zero actual coverage if almost all items
in the database are within the uncovered 50% of the bounding box.

Accordingly, the desired coverage here is only used to generate a wide range of queries, and no correlation with actual coverage is assumed. Instead, the actual coverage of queries is measured precisely by executing the queries on the database. The random query generation process continues until a sufficient number of queries is found which have the desired coverage.
Bibliography


[40] Jiawei Han, Micheline Kamber, and Jian Pei. Data Mining: Concepts and Techniques. Elsevier, 2006.


