A Framework for Soft Real-time Analysis in OLAP Systems

Omer Baluch

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fullfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

March 18, 2012

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Omer Baluch

Entitled: A Framework for Soft Real-time Analysis in OLAP System.

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Sabine Bergler

_____ Examiner
Dr. Gregory Butler

_____ Examiner
Dr. Brigitte Jaumard

_____ Supervisor
Dr. Todd Eavis

Approved by _____
Chair of Department or Graduate Program Director

_____
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Date _____

ABSTRACT

A Framework for Soft Real-time Analysis in OLAP Systems

Omer Baluch

OLAP systems are designed to quickly answer multi-dimensional queries against large data warehouse systems. Constructing data cubes and their associated indexes is time consuming and computationally expensive, and for this reason, data cubes are only refreshed periodically. Increasingly, organizations are demanding for both historical and predictive analysis based on the most current data. This trend has also placed the requirement on OLAP systems to merge updates at a much faster rate than before.

In this thesis, we propose a framework for OLAP systems that enables updates to be merged with data cubes in soft real-time. We apply a strategy of local partitioning of the data cube, and maintain a "hot" partition for each materialized view to merge update data. We augment this strategy by applying multi-core processing using the OpenMP library to accelerate data cube construction and query resolution.

Experiments using a data cube with 10,000,000 tuples and an update set of 100,000 tuples show that our framework achieves a 99% performance improvement updating the data cube, a 76% performance increase when constructing a new data cube, and a 72% performance increase when resolving a range query against a data cube with 1,000,000 tuples.

ACKNOWLEDGEMENTS

I would like to thank Dr. Todd Eavis, whose guidance was invaluable in completing this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Organizations all over the world are collecting data at an unprecedented pace and storing it in large databases. It is no longer unusual to hear of terabyte size databases, and there are documented examples of databases that have exceeded a petabyte [68]. As an example, in 2009, Yahoo! [3] reported having over 6 petabytes of data in their Everest system and expected to pass the 10 petabyte threshold by the end of that same year.

The data is collected from diverse sources such as retail sales transactions, online browsing activity, social networking sites, scientific research, or even sensors [6][26] [47]. Much of this data is managed and stored using an online transaction processing system (OLTP). Relational database management systems (RDBMS) [49] are an example of OLTP systems, and are primarily designed to handle a large volume of small transactions efficiently. The data collected by these systems is described as *primitive data* because the main purpose of these systems is to keep a record of transactions, and not to analyze the collected data [43]. Current RDBMS are based primarily on the relational model described by Codd [14].

As the size of databases has grown over the years, organizations have been eager

to analyze the data to extract information from it that will assist them in their decision making process. One method is to copy the data from the RDBMS into a data warehouse (DW) system , and then transform it using a decision support system (DSS) [20]. The transformed data produced by a DSS is refered to as *derived data* because it is used to make management decisions, or to derive an answer to a question based on certain facts [43]. A few examples of DSS are Business Intelligence (BI) tools that include online analytical processing (OLAP) tools, data mining software, and expert systems.

A DW is considered to be a historical snapshot of the operational database, and is a separate entity from the operational database [49]. Traditionally, the data from the RDBMS was loaded into the DW either weekly or even nightly, usually when the RDBMS was offline. The reason for this is that the data needs to be transformed through a process called ETL (Extract-Transform-Load), so that data from hetero-geounous RDBMS sources is made uniform in the DW. Due to the high volume of data that needs to be transformed, the enormous burden on the operational database made it difficult to keep the RDBMS system online and at the same time upload the data into the DW [50].

OLAP systems are designed to quickly answer multi-dimensional queries against large DW systems. Based on the multidimensional *data cube* data structure [38], aggregrate values such as sum and averages are pre-computed using the data in the DW, and stored in the data cube. Users retrieve the values by locating the intersection of the specified dimensions. Constructing data cubes and their associated indexes is time consuming and computationally expensive [12]. For this reason, data cubes are only refreshed periodically.

Increasingly, organizations are demanding for both historical and predictive analysis based on the most current data. In order to respond to this demand, DW technologies have moved towards real time updates. Changes in the RDBMS are being pushed into the data warehouse at a much faster rate than before. Instead of weekly or even daily updates, changes from the transactional system are being continuously pushed into the data warehouse [5]. This trend has also placed the requirement on OLAP systems to merge updates at a much faster rate than before.

Sidera is a parallel OLAP server developed by Eavis that runs on a commodity cluster [29]. It exploits the clustering properties of the Hilbert space filling curve to achieve balanced data cube partitioning across the nodes in the cluster. Indexes are built using the R-tree data structure [28][40] and both the data and indexes are compressed using Hilbert Tuple Differential Coding(HTDC) [27]. The HTDC compression allows up to 90% compression of the distributed datacube, as well as up to 98% compression for the R-tree indexes [27]. Sidera also supports multi-dimensional caching, hierarchy-aware query resolution, and approximate query answering [30][31][32].

HTDC compression is a computationally expensive operation. Because complete materialized views must be decompressed before merging updates, the run-time for updating the data cube is almost as high as that for building the data cube. Query resolution times also suffer because blocks containing matching tuples must be decompressed in order to retrieve the tuples. Thus, the information contained in the data cubes quickly looses its effectiveness because updates cannot be merged fast enough without affecting system performance and user experience.

In this thesis, we propose a modified architecture for the Sidera OLAP server that enables updates to be merged with the data cube in soft real-time. We apply

a strategy of local partitioning of the data cube, and maintain a "hot" partition for each materialized view to merge update data. We augment this strategy by applying multi-core processing to the HTDC compression and decompression stages in both the indexer and query engine. This allows us to reduce the time required to merge updates and therefore increase the frequency at which the data cube can be refreshed, while improving the run-times for building the data cubes and query resolution. As far as we know, no research has yet explored a local partitioning strategy and multi-core processing in an OLAP server that uses HTDC compression such as Sidera to achieve soft real-time updates.

The rest of this chapter provides an overview of the work done to implement soft-real time updates in Sidera, and explains how the thesis is organised.

1.1 Overview of Research

The research presented in this thesis focuses on the modifications done to the architecture of the Sidera OLAP server to allow it to merge updates in soft real-time. More specifically, the following changes were applied to the indexer and query engine, which are the two key components of Sidera that generate, update and query the data cube.

1.1.1 Local Partitioning of the Data Cube

We implemented local partitioning of the data cube and compressed each partition using HTDC compression. Each partition has an Linear Breath First (LBF) R-tree index [28] which is also compressed. A partition table is used to keep track of the number of partition per materialized view. We implemented the indexer to run on

a single node in order to obtain measurable results. We also used Hilbert packing to distribute the tuples across the partitions. This minimized the number of blocks retrieved from disk when resolving a user query.

1.1.2 Using a "hot" Partition for Fast Merging of Updates

For each materialized view, a "hot" partition is used to merge updates. As the "hot" partition reaches a certain size, it is capped and a new "hot" partition is started. Since merging updates is a computationally expensive operation, the "hot" partition was limited to a certain size. A side effect of merging updates with a "hot" partition is the introduction of duplicate tuples. For this reason, the query engine was modified to remove duplicate tuples and aggregate measures on the fly.

1.1.3 Using Multi-core Processing

Exploiting the processing capabilities of multi-core processors, we implemented shared memory parallelism in both the indexer and the query engine to handle computationally expensive operations such as Hilbert conversions, compression and decompression, and sorting. The OpenMP library was used to incrementally add multi-core processing to the Sidera code base.

An analysis of the Sidera codebase showed that certain time consuming operations consisted of a large number of manipulations of tuple data. Transformations such as converting multi-dimensional tuples to Hilbert indexes can be done independently from one another, which made them ideally suited for parallel processing. Using the pre-processor directives supplied by the OpenMP library, we divided these repetitive operatons into disjoint sets and distributed the work across multi-cores. The OpenMP library was also used to control the number of threads available to our system, as well

as general thread management such as initialization and cleanup.

1.1.4 Replacing the Buffering Subsystem

The original Sidera indexer and query engine include a buffering subsystem that uses a series of intermediate files when converting data from one form such as multidimensional tuples to another form such as Hilbert indexes. This adds significant I/O overhead to both the indexer and the query engine, and coupled with overlapping I/O read and write calls, makes it difficult to parallelize on a single disk system.

We replaced the original file based buffering subsystem with an in-memory buffering subsystem. Materialized views are read sequentially into memory, and all intermediate transformations are done in memory and are fully parallelized. When the transformations are completed, the results are written back sequentially to disk, minimizing read and write calls, as well as the number of disk seeks.

1.2 Thesis Organization

The thesis is organized as follows

- *Chapter 2* explains the key concepts behind the Sidera OLAP server as well data partitioning and multi-core processing. These will serve as a basis to understanding the ideas presented in the thesis.

- *Chapter 3* begins with a brief overview of the Sidera architecture, followed by a more detailed description of the indexer and the query engine. For each component, we describe the original algorithms, and then describe the changes we made to achieve soft real-time updates. This includes implementing the local partitioning strategy, maintaining a "hot" partition for merging updates,

and adapting the query engine to aggregate duplicate records on the fly. The chapter also explains how we replaced the file based buffering subsystem with an in-memory based buffering subsystem, and how the indexer and the query engine were modifed to run on a single system with multiple multi-core CPUs.

- *Chapter 4* describes the conditions under which we tested the modified architecture and discusses the results we obtained by running the testcases.

- *Chapter 5* presents our conclusions based on the results we obtained in Chapter 4. We also provide a discussion on the limitations we encountered during our implementation, and suggestions for future work based on our research.

Chapter 2

Background

2.1 Introduction

Decision support systems (DSS) have been around for several decades and are used to transform data into information with the eventual goal of supporting some type of decision making process within an organization.

Business Intelligence (BI) is a term used to describe a type of *data-driven* DSS that focuses on doing historical and predictive analysis of business data in order to support corporate decision making.

BI systems are assembled using several components that allow accurate and quick processing of large volumes of data, and presenting the results to the users in a timely and easy to understand fashion [20]. Central to the BI architecture are the data warehouse (DW), Online Analytical Processing (OLAP) tools, and the BI presentation layer. Figure 2.1 illustrates the BI architecture.

In a competitive world, making decisions based on the most recent available data has become increasingly important. There is a growing demand by organizations for BI tools that generate reports based on the most up-to-date operational data. In this thesis, we describe the modifications we made to an OLAP server that uses Hilbert

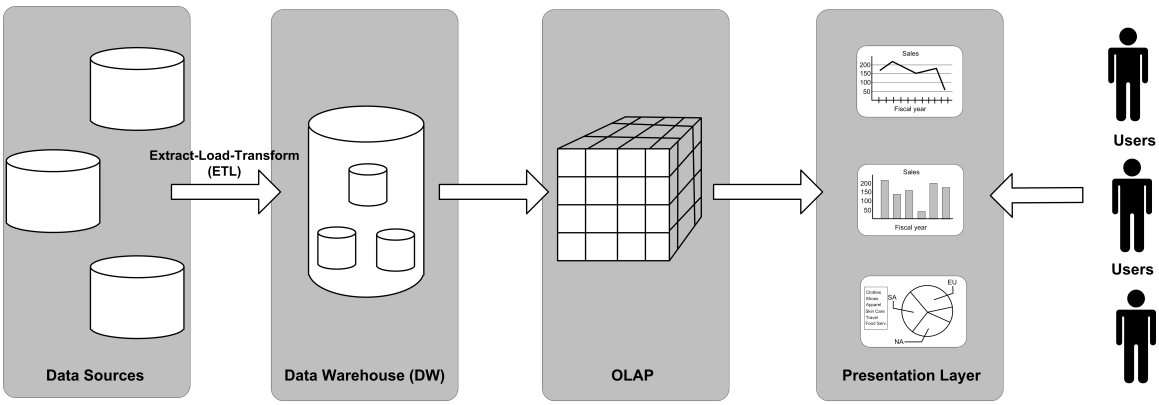


Figure 2.1: The Business Intelligence (BI) Architecture.

Tuple Differential Coding (HTDC) compression to enable updates to be merged in soft real-time without decreasing system performance.

The rest of this chapter presents the concepts necessary to understand the solution proposed in this thesis. We begin with a discussion on DW and OLAP, specifically maintaining OLAP data cubes. Next, we continue with a discussion on Hilbert space filling curves, Hilbert tuple differential coding (HTDC) compression, packed Hilbert r-trees, and data partitioning strategies. Finally, we provide an overview of parallel programming, multi-core programming and soft real-time systems.

2.2 Data Warehousing

Data warehouses (DW) are the foundation upon which many DSS are constructed. Primitive data from heterogenous Online Transactional Processing (OLTP) systems is collected, transformed and stored in a DW. Aside from BI tools, DW systems are also used by other DSS such as data mining tools and expert systems.

A key feature of DW is how the data is structured. Instead of following Codd's

rules [14] for table normalization, data in a DW is stored in a large *fact table* and a limited number of *dimension* tables provide context to the fact table. This structure is refered to as a *star schema*. Figure 2.2 shows an example of a star schema. A more normalized structure called the *snowflake schema* is also sometimes used.

There are several reasons why BI tools do not execute directly using data from OLTP systems. The most important one is peformance. OLTP systems are designed primarily to handle a large volume of queries that affect a small number of records, whereas BI tools usually deal with a small number of queries that process a large volume of records or data [1]. Therefore, running BI queries against the OLTP systems result in degraged performance [12].

New data from the OLTP system is merged periodically with the DW. As the fact table becomes larger over time, merging new tuples and rebuilding indexes become increasingly expensive. For this reason, new data from the OLTP system is normally pushed into the DW only monthly, weekly or daily [48][64]. This implies that data in the DW never represents the most current data in the OLTP system, and users running reports on top of the DW warehouse using OLAP tools will almost never obtain information based on the most recent snapshot of the OLTP system. We say that the data in a traditional DW is *stale* with regards to the OLTP system. Several strategies have been proposed to increase the update frequencies between the OLTP system and the DW.

Scheduling strategies have been proposed to minimize the DW staleness [37][5][36]. Scheduling algorithms trigger a refresh cycle when the fact table reaches a certain staleness threshold. Along the same lines, [51] mentions a strategy to track the staleness of the data in the DW with respect to the performance degradation if the

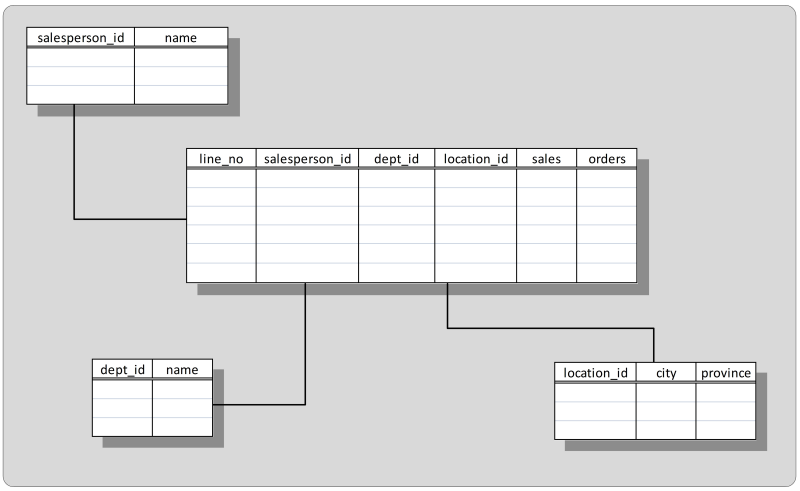


Figure 2.2: The Star Schema.

DW was synchronized with the OLTP.

Others have looked at methods to allow updates to be incrementally merged with the DW while it remains online. In [63], using table replication and temporary tables was proposed to speed up merging update data.

Finally, another strategy is parallel DW. According to [48], the Extract-Transform-Load (ETL) process, which assembles data from heterogenous sources into the DW, consumes up to 70% of the processing time in a DW. In [70], the authors explain how the ETL process offers the possibility to implement both task and data parallelism and proceed to provide a framework for parallel ETL programming. Their results show that by investing a few more cpu cycles to implement their framework, it can significantly reduce the wall-clock time of an ETL process.

2.3 OLAP

Online analytical processing (OLAP) tools allow multi-dimensional queries against a DW to be answered quickly. This is done by pre-computing aggregates and storing

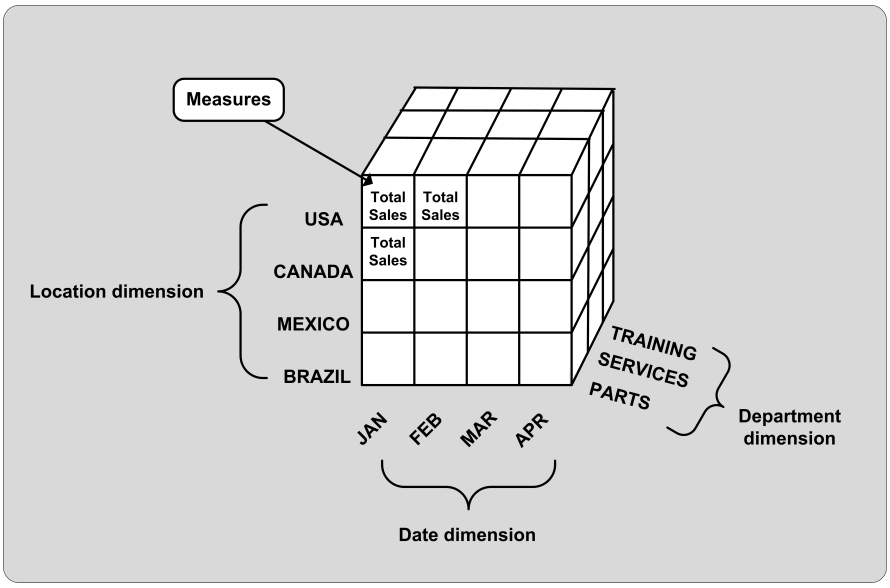


Figure 2.3: An OLAP Data Cube with 3 Dimensions.

the results in a data structure called the *data cube* [38]. Figure 2.3 shows a data cube with 3 dimensions. Given that a fact table in a DW has d dimensions $\{A_1, A_2, A_3, ... A_d\}$, then pre-computing the aggregate for every dimension combination results in 2^d *cuboids*. Together, these cuboids constitute the data cube. We also refer to these cuboids as *materialized views*, or simply as *views*.

User queries are resolved quickly by identifying the view that contains the pre-computed aggregate value for the specified dimensions and retrieving the value from the view. For a DW with a very large fact table, using a materialized view is faster than computing the aggregate values at query time.

OLAP tools have presented several challenges since their appearance. Data cubes require additional storage. For example, a DW fact table with 8 dimension tables will generate $2^8 = 1024$ cuboids that must be stored on disk. Furthermore, certain dimension combinations produce views that are almost or even completely empty

leading to wasted storage space because space must still be allocated for these views [67].

Cube construction is also a computationally expensive and time consuming operation. Like the DW, data cubes are traditionally refreshed monthly, weekly, or even daily, leading to the same staleness problem as in the DW.

An important component of OLAP servers are indexes, especially if the data cubes are stored on disk. R-trees or structures derived from the R-tree have emerged as a popular choice for multi-dimentional indexing [40][62][65][7][45][46][28]. We provide more details on a variation called the packed Hilbert R-tree data structure a little further in this chapter.

Parallel OLAP servers have been constructed on commodity clusters. These partition the data cube across the nodes in the cluster thereby distributing the work load among multiple machines [24][21][22][23]. Chen *et al.* [72][10][13] have demonstrated how large ROLAP data cubes can be constructed in parallel. These OLAP servers are built using a *shared nothing* parallel architecture because it has been shown to be more scalable compared to the more complex *shared everything* architecture. Furthermore, most of this research was done at a time when commodity computers were generally comprised of a one single core processor.

Other research has focused on strategies that minimize the number of materialized views that need to be constructed as well as the space required to store them on disk. Dehne *et al.* [25][23] have demonstrated that a partial data cube can be constructed and outlined parallel algorithms that reduced the cost of constructing the date cube from 20% to 70% compared to using naive methods. A partial data cube means that only a subset of the 2^d materialized views need to be constructed and stored.

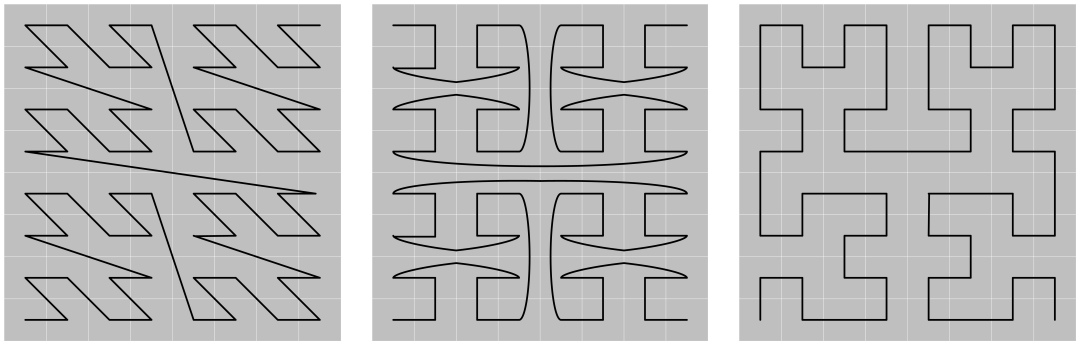
Sismanis *et al.* [67][66] have proposed the Dwarf cube, a highly compressed structure that stores pre-computed aggregate values and allows cube manipulation such as querying, rollup and drilldown. They have succeeded in compressing a petabyte cube to 2.3GB in 20 minutes, resulting in a 1:400000 reduction in size. Finally, Eavis and Cueva [27] have proposed a framework for compressing data cubes and their indexes using Hilbert Tuple Differential Coding (HTDC) compression, achieving compression rates of more that 90% for the data cubes, and up to 98% for their associated packed Hilbert R-tree indexes.

2.4 Hilbert Space-filling Curve

A space filling curve is a non-differentiable curve whose range [0,1] passes through every point in an N-dimensional space. It can also be described as a continuous, bijective function that maps every point from a unit interval [0,1] to a unique point in an N-dimensional unit hypercube $[0,1]^N$. The first space filling curves were described by Giuseppe Piano and space-filling curves that map a unit interval to a 2-dimensional space $[0,1]^2$ are known as Peano curves [59]. Peano curves are also fractals with a fractal dimension of 2.

We focus on two properties about space filling curves that are important to the material presented in this thesis. First, the transformation from a N-dimensional hyperspace to a space filling curve is said to apply a *total order* to the points in the hyperspace. This means that the transformation is *reflexive*, *antisymmetric*, *transitive*, and *trichotomous*.

Second, space filling curves preserve the locality of multi-dimensional points when they are mapped to the single dimension curve. This means that points that are



(a) Z-order Curve (b) Gray Code Curve (c) Hilbert Curve

Figure 2.4: Examples of Space Filling Curves.

spacially close to each other in the N-dimension hypercube are mapped to points on the space filling curve that are close to each other. The level of clustering varies depending on the space filling curve [54].

There are many different space filling curves. Three important space filling curves are the Z-order curve [57], Gray code curve based on the Gray code [41], and the Hilbert curve [42]. An example of each are shown in Figure 2.4.

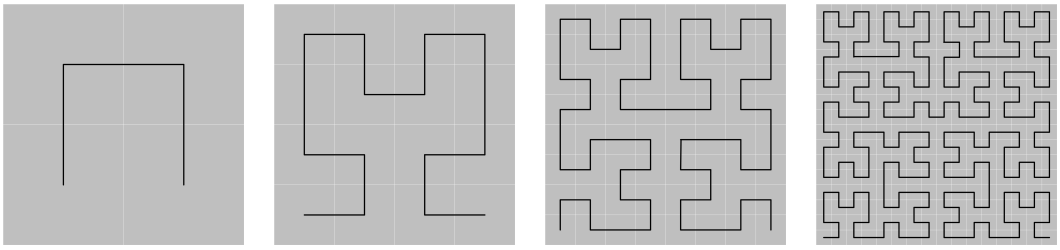
Among space filling curves, the Hilbert curve has been demonstrated to have superior clustering properties over other curves such as z-order and Gray code [44][54].

Moon *et al.* [54] provide the following notation 2.1 to describe successive iterations of Hilbert curves:

$$\mathcal{H}_k^d : [0, 2^{kd} - 1] \rightarrow [0, 2^k - 1]^d \tag{2.1}$$

where d is the number of dimensions in the hyperspace and k denotes the k^{th}-order (or iteration) of the curve. Figure 2.5 shows the first four orders (or interations) of the Hilbert curve.

In Sections 2.5 and 2.6, we explain how these two properties of Hilbert space filling



(a) $\mathcal{H}_1^2$ (b) $\mathcal{H}_2^2$ (c) $\mathcal{H}_3^2$ (d) $\mathcal{H}_4^2$

Figure 2.5: First Four Orders of the Hilbert Curve.

curves are used to used to achieve block level compression for the OLAP data cube and indexes, and to construct R-tree indexes that minimize the number of disk blocks accessed during query resolution.

2.5 Hilbert Tuple Differential Coding (HTDC)

Hilbert tuple differential coding (HTDC) is a form of compression that is based on the Hilbert space curve that allows the data cube as well as the R-tree indexes to be compressed. This reduces the storage space required for both the data cube and the indexes.

HTDC relies on the total ordering imposed when points from a d-dimensional space is transformed to a Hibert space filling curve of length s_d. It was described by Eavis and Cueva [27], and it is a variation of the technique called Tuple Differential Coding described by Ng and Ravishankar [58]. The following is a summary of the technique.

Assume a relation $\mathcal{R} =< A_1, A_2, ..., A_d >$, where A_i is a dimension with cardinality $|A_i|$ for $i = \{1, 2, 3, ..., d\}$. $\mathcal{R}$ represents a multi-dimensional space where the dimensions are used to specify the points in the space, and an individual tuple $\varphi(a_1, a_2, ..., a_d)$ is a point in the space. The transformation proposed by Ng and

Ravishankar imposes a lexicographic ordering to the tuples in $\mathcal{R}$ by the mapping function:

$$\varphi : \mathcal{R} \rightarrow \mathcal{N}_{\mathcal{R}} \tag{2.2}$$

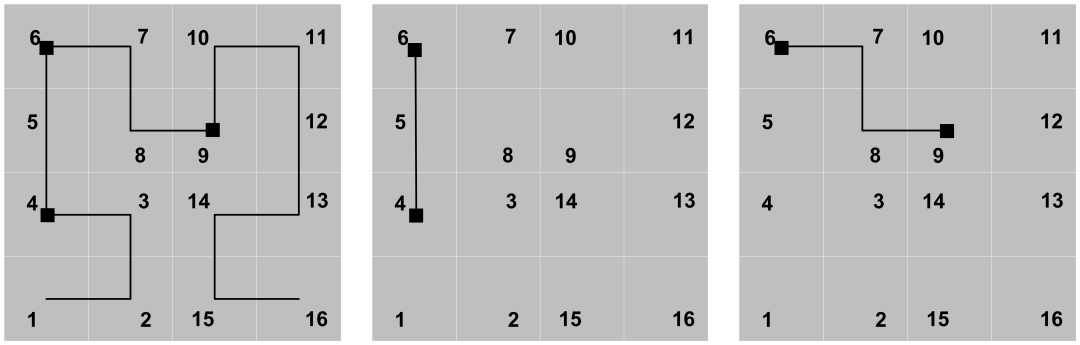
where $\mathcal{N}_{\mathcal{R}} = \{0, 1, ..., \|\mathcal{R}\| - 1\}$ and $\|\mathcal{R}\|$ is defined as follows:

$$\begin{aligned} \|\mathcal{R}\| &= \prod_{i=1}^{d} A_i : \varphi(a_1, a_2, ..., ad) \\ &= \sum_{i=1}^{d} (a_i \prod_{j=i+1}^{d} |A_j|) \end{aligned} \tag{2.3}$$

Once the tuples have been transformed to their ordinal values using the mapping function 2.2, the ordinals are sorted in ascending order. The compression process starts by storing the first ordinal in the first block, and then storing the differences between subsequent tuples. This is defined as the difference between tuples $\varphi_i - \varphi_{i-1}$, where $0 < i \leq |\mathcal{R}|$. The differences are saved in the block using run length encoding (RLE). Since differences are small and require a small number of bits to encode them, Ng and Ravishankar have reported a 68% compression ratio. Note that the compression ratio for larger relations will be higher since the differences between tuples will be smaller, leading to smaller number of bits required to save the differences.

Eavis and Cueva proposed a similar technique, but where the mapping function applies a Hilbert order on the ordinals produced in the transformation stage. If we assume a space with d dimensions, we can transform it to

$$C_{HIL} : \{1, 2, ..., s^d\} \rightarrow \{1, 2, ..., s\}^d \tag{2.4}$$



(a) $\mathcal{H}_1^2$ (b) $\mathcal{H}_2^2$ (c) $\mathcal{H}_3^2$

Figure 2.6: First Four Orders of the Hilbert Curve.

where a single multidimensional tuple $\varphi(a_1, a_2, ..., a_d) = C_{HIL}(a_1, a_2, ..., a_d)$.

Figure 2.6(a) shows a 2 dimensional Hilbert space where each dimension has a cardinality of 4. This space maps to 16 unique values on the Hilbert space filling curve. As an example, assume three tuples in the relation $\mathcal{R}$ with the tuples $C_{HIL}(1, 2)$, $C_{HIL}(1, 4)$, $C_{HIL}(3, 3)$. These are shows as points on the Figure 2.6(a).

The first step would be to transform the three tuples to a Hilbert ordinal. Again, looking at Figure 2.6(a), $C_{HIL}(1, 2)$ maps to 4, $C_{HIL}(1, 4)$ maps to 6, and $C_{HIL}(3, 3)$ maps to 9. After this, we sort the Hilbert ordinals in ascending order.

The first value stored in a compressed block will be 4. This is stored as a full integer value, requiring 32 bits on most platforms. For the next value, we calculate the different $\rho = 6 - 4 = 2$. Since we can express 2 as 10 in binary format, we remove all leading zeros and use only 2 bits to store the difference, saving 30 bits if we assume that an integer would require 32 bits. The next difference is calculated $\rho = 9 - 6 = 3$. Again, 3 is 11 in binary format, and we only need 2 bits and we save 30 bits. If we were to store multidimensional values, then we would need to store the (x,y) values of the coordinates, requiring 2 x 32 = 64 bits. So for the first tuple, which is stored

uncompressed, we save 32 bits since we only store one value. For the differences, we save 62 bits for each coordinate. Eavis and Cueva have reported compression ratios of more than 90% for data files, and up to 98% for the indexes.

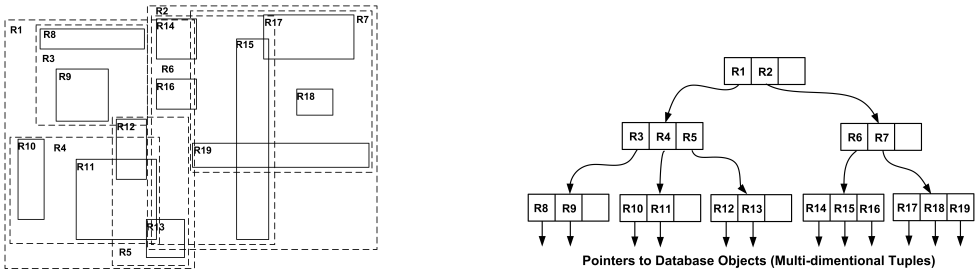
Since the Hilbert transformation is a bijective function, compressed blocks can be decompressed to retrieve the original multidimensional tuples. Furthermore, since the first tuple in each disk block is stored uncompressed, decompression can be done at the block level. When resolving a user query, only the blocks containing tuples that satisfy the query parameters need to be retrieved and decompressed.

2.6 Hilbert R-tree Indexing

The R-tree structure was first proposed by Guttman [40] as a solution for indexing topological and cartological databases. The problem was that geographical locations usually have at least two coordinates, and structures such as B-trees and ISAM indexes were inadequate because they ordered keys along a single dimensions. Multidimensional indexes such as Quad-tree [33], k-d trees [8], or the K-D-B tree [60] were also not sufficient because they either assumed a static database, which means that it was difficult to merge updates, or they performed poorly on range queries. The following is a quick overview of the R-tree.

In the R-tree, a leaf node is identified as $(I, \textit{tuple-identifier})$ where $\textit{tuple-identifier}$ is a pointer to an object stored the database, and I defines a d-dimensional bounding box that encloses the database object. We can also say that $I = (I_1, I_2, ..., I_d)$ where I_i holds the maximum and minimum value of that object for the dimension i.

Non-leaf nodes are constructed by adding entries in the form $(I, \textit{child-pointer})$, where *child-pointer* is a pointer to either a leaf node, or to another non-leaf node.



(a) Bounding Boxes. (b) After Construction.

Figure 2.7: The R-tree.

I defines the d-dimensional bounding box that encloses every bounding box in the child node.

Using Figure 2.7(a), we illustrate how an R-tree is constructed recursively from the leaf level up to the root node. Rectangles R8 to R19 are leaf-nodes and point to multidimensional tuples in the database. We start by constructing the non-leaf node R3, which encompasses nodes R8 and R9. We continue with node-leaf node R4, which encompasses nodes R10 and R11. We continue this process until we have grouped all the leaf level rectangles into the non-leaf rectangles R4 to R7. We repeat this process and regroup R3 to R5 inside R1, and nodes R6 and R7 inside R2. Finally, R1 and R2 are grouped inside the root node. Figure 2.7(b) shows the R-tree after construction. Searching for an object or a range of objects is accomplished using a depth-first search algorithm starting with the root node.

Starting with the assumption that most multidimensional databases are static, Rossopoulos and Leifker [62] provide a method for a "packed" R-tree so that there is minimal number of bounding box overlap and thus reduce dead-space. Instead of maintaining between m and M number of entries per non-leaf nodes, their method fills every node with M rectangle. This produces a very shallow but broad R-tree and

improves search queries. This is done without compromising the dynamic nature of R-tree.

Faloutsos and Kamel [45][46] push this idea further and first apply Hilbert ordering to the multidimensional database object before packing and constructing the R-tree. The clustering properties of the Hilbert space filling curve describe in Section 2.4 ensure that database objects that are spatially close together will be grouped in the same non-leaf node. They report better perfomance especially for skewed data distribution that better reflect real world situations.

The depth first search algorithms used for searching R-trees does not specify the order in which matching blocks are read from disk, and can lead to significant disk trashing due to high number of seeks. Eavis and Cueva [28] introduce the LBF R-tree in , including a linear breath first search (LBFS) algorithm that guarantees that blocks are read sequentially from a file, and degrades to a linear scan of the database in the worst case situation. Their algorithm minimizes the number of seeks a disk must perform, which is one of the major factors contributing disk latency. Accordingly, the LBF R-tree has been shown to reduce disk seeks by more than 50% compared to other R-trees.

2.7 Data Partitioning

Database partition has been a feature of many commercial databases such as IBM DB2 [16], Oracle [19] and SQL server [18]. Data partitioning means that the data in a database or even just a table is separated into different disjoint subtables. Partitioning tables helps manage the tables more easily and also speeds up performance in environments where the partitions can be manipulated in parallel.

Agrawal *et al.* [2] describe two major ways of partitioning database tables.

- **Horizontal Partitioning** breaks tables across rows using a partitioning strategy such as range partitioning or hash partitioning. The rows may still reside in one single table, but a separate index is then built for each partition. *Sharding* is a technique where a database table is horizontally partitioned and each partition resides on a separate logical or physical server. Each server has a copy of the database schema, and no longer needs to be aware of the other servers.

- **Vertical Partitioning** breaks tables into disjoint sets of columns, and each set is assigned to a subtable. The only column that is repeated in every subtable is the *key* column.

2.8 Parallel Computing

Parallel computing allows multiple program instructions to be executed simultaneously on separate processing units. This is done by dividing instructions and data into discrete units and then distributing them across the processing units for execution. Parallel systems include various different architectures such as computers each with a single core processor connected together using a network, a single computer with multiple processors connected together by a bus, or a combination of both. More recent architectures also include processors that contain multiple processing units called cores in a single chip.

Physical limits such as power consumption and heat dissipation have placed a limit on the developement of single core processor architectures, leading processor manufacturers to switch to multi-core architures [17]. Cost per instructions is lower

for multi-core processors because they can execute more instructions at lower clock frequencies, which lowers the power consumption.

A common method of classifying parallel architectures is based on Flynn's taxonomy [35][34], which was introduced in the 1960s. It divides the architecture based on Instructions and Data, either of which can be Single or Multiple. Another common method for classifying parallel architectures separates parallel systems into two distinct groups. The following is a brief description of the two groups.

- **Shared-memory** systems, which are also refered to as *"shared-everything"* systems, have multiple processing units that access the same global address space and connect to the same memory using a shared bus. Any change to a memory location affects all processors. We also define shared-memory systems as having either *Uniform Memory Access (UMA)* or *Non Uniform Memory Acess (NUMA)*, depending on how the memory is organized in the system. In the first case, each processor can access any memory location with the same latency. In the second, each processor can access certain memory locations faster than other locations.

  An advantage of shared-memory systems is that data sharing is fast and uniform. Disadvantages include problems with scalability because traffic on a shared bus ncreases geometrically with the number of processing cores. Cache coherency also adds complexity to this architecture because each processing unit has access to a private cache which is not always synchronized with the shared memory. When a processing carries out an instruction that affects a shared variable, another processing unit may attempt to change the same variable before the cache value of the first processing unit is written back to memory. This

is more apparent in NUMA architectures. We define a NUMA architecture that can handle cache coherency as cache coherent NUMA, or cc-NUMA [11].

- **Distributed memory** systems, which are also refered to *"shared-nothing"* systems, have multiple processing units that each have access to their private global address space. The processing units must use a message passing protocol to communicate with each other. This architecture is more scalable than shared-memory systems, and is easy to assemble using commodity off-the-shelf processors and an Ethernet network with TCP/IP for communications. Another advantage of these systems is that there is no need to maintain cache coherency since each processor has exclusive access to its memory.

2.9 Parallel Programming Models

Modern processors implement techniques such as bit level and instruction level parallelism that enable more instructions to be executed per clock cycle, but in order to take full advantage of a parallel computer system, developers must explicitely design their applications to run in parallel. There are two important types of parallelism.

- In **task parallelism**, separate tasks or functions can be executed on separate processing units. One of the challenges of task parallelism is how to distribute work evenly across processing units.

- In **data parallelism**, a large data set is partitioned into smaller disjoint sets such that each set can be processed independently from the other. These can then be distributed to separate processing units and the same instructions are carried on each disjoint set. An example would an operation on a matrix.

Designing and implementing parallel programs is difficult and often result in bugs that are difficult to detect because these bugs are related to thread timing. Programmers must also manage access to shared resources (critical sections) and must also coordinate communication between the various executation cores. For this reason, several parallel programming models have been developed over the years to abstract many of the painful tasks when programming parallel applications.

- **Shared-memory** model (threads model)

  In this model, the programmer is presented with a uniform memory model, although physical memory could be located on separate physical machines. Parallel execution is achieved by using threads which execute instructions in parallel. Threads access the same memory, and the programmer is responsible for ensuring that data is not corrupted or incorrect because of race conditions, and against deadlocks when using locks and semaphores. APIs built on this model are Posix threads [9], OpenMP [69], Unified Parallel C [71], X10 [15] and OpenCL [39].

- **Message-passing** model (distributed memory model)

  In this model, the programmer is presented with a separate memory model for each processing unit, and the programmer must use a message passing protocol to exchange information between the processing units. The Message Passing Interface (MPI) API [53] is now the de-facto library used for message passing when developing applications for distributed memory systems. It is still the responsibility of the programmer to identify the parallel components, and to structure the application to run on parallel systems. In many cases, APIs such

as OpenMP and MPI can be mixed to create applications that take advantage of both models.

2.10 Evaluating Parallel Applications

Simply running a sequential application on a parallel system with p processing units does not always results in p times speedup. The application must first be restructured so that instructions are distributed across the processing units. We define the speedup achieved as follows [52]:

$$\text{Speedup} = \frac{\text{Execution time on 1 processor}}{\text{Execution time on } p \text{ processors}} \tag{2.5}$$

The top part of the expression (2.5) represents the sequential execution time and the bottom part the parallel execution time. Under ideal conditions, we would like the parallel execution time to be equal to $\frac{1}{p}\times$ sequential execution time. In other words, we would like the speedup to be equal to p. This means that the more processor we allocate to the application, the faster it will execute.

Two factors limit the speedup we can achieve. The first factor is the ratio between the time a program spends executing code that is inherently sequential, i.e. cannot be executed in parallel, and executing code that executes in parallel. The second factor that affects speedup is the the overhead associated with managing parallel applications such as thread managements and communication between executing units.

To compute the speedup achieved when using p processors to execute a parallel

application, Amdahl proposed an expression which is now known as Amdahl's law [4]. After profiling an application, we define S as the fraction of time it spends executing code that cannot be executed in parallel, and $(1-S)$ as the fraction of time it spends executing code that can be execute in parallel. Then, Amdahl's law defines speedup on p processors as follows:

$$\text{Speedup} = \frac{1}{S + \frac{(1-S)}{p}} \tag{2.6}$$

Amdahl's law provides a close estimate to the speedup achieved when a parallel application is run on p processors. It can also be used to estimate an upper bound on the speedup that can be achieved. Assume an infinite number of processors are allocated to the application, then

$$\text{Speedup} = \lim_{p \to \infty} \frac{1}{S + \frac{(1-S)}{p}} = \frac{1}{S} \tag{2.7}$$

This means that the maximum speedup that can achieved when running an application on a parallel system is related to the fraction of time is spends running instructions that can only run sequentially.

2.11 Multi-core Programming

Back in 1965, Moore [56] predicted that the number of components in an integrated circuit roughly doubled every two years. This is known as Moore's law, which states that the processing speed and the storage capacity of computer systems doubles every 18 months. Processor development has adhered to this trend by increasing the number

of transistors on a single chip and by reducing the size of the transistors allowing more transistors to be packed per unit area. By the end of the 1990s, cpu manufacturers had realized that physical limitations such as power usage and heat dissipation made it increasingly difficult to pack more transistors per unit area. In order to continue increasing performance, the focus shifted to multi-core cpu architectures. These are single chips with multiple executing units or core packed together. This has allowed the net number of intructions executed to be increased by executing them in parallel. The processor itself could run at a lower frequency, but the effective speed was the sum of the instructions run on all the cores.

There are several ways for a system to take advantage of the multiple cores available. The first way is for the operating system to schedule a separate application to run on each core. For example, while a user is encoding video using one core, the virus scan application can run on the other core. In single core cpus, the operating system would have to context switch and allocate a time slice for each of these applications. The other way is if an application process has multiple threads, then two or more threads can run on separate cores. But in order for this happen, it is up to the application to create the threads and ensure that resources are accessed appropriately. This means that the application developer must explicitely design the application to be multi-threaded.

2.12 OpenMP

OpenMP stands for Open Multi-Processing and it is a multi-platform shared memory multi-programming API that allows application developers to implement multi-threaded applications without directly using libraries such as Pthreads. In fact,

OpenMP use Pthreads behind the scenes, but abstracts a lot of details away from the user. This is done by providing the programmer with a set pre-precessor directives that allow multiple threads to be spawned when sections can be processed in parallel. There are also directives that allow to define shared variables and also the private variables for the separate threads. When a parallel section is completed, the API takes care of either removing the threads, or by returning them to the thread pool, where they can be recycled for another parallel section.

The API also provide contructs for declaring barriers and other synchronization points, as well as allowing the programmer to decide the number of threads to use. It follows a model where the main thread can fork multiple slave threads to do work on a parallel section before joining back to the main thread. This is a classic example of the fork-join model. Slave threads can either do the same work but each on a different part of the data (data parallelism), or each execute a separate task (task parallelism).

OpenMP is used to parallelize code across multiple cores on a single cpu, or across multiple cpus but on a machine or a cluster that runs a single OS. It does not distribute work across independant machines, like MPI does. For this reason, it is not uncommon to see OpenMP used alongside MPI in many parallel and distributed applications.

OpenMP has many advantages over Pthreads. The API is much simpler and makes it easier to implement multi-threading without the complexity of Pthreads. The pre-processor directives can be added to non-multithreaded programs to convert them to multi-threaded programs without having to re-write the structure of the program. This is referred to incremental parallelism. When adding Pthreads to a non-parallel program, the structure must often be re-written. Platforms with non

OpenMP capable compilers can still compile the code because the compiler directives are simple flagged as warnings. The binary that is produced executes the parallel sections using a single thread.

2.13 Soft Real-time Updates

Real time computing consists of software and hardware that perform a task and can guarantee a certain deadline. We can define two different kinds of real-time computing, based on what happens if the deadline is not met.

Hard: if the deadline is not met, this leads to a system failure because the deadline is mission critical. As an example, certain critical subsystems in a nuclear plant must repond to changes within a determined period of time. Failure do so may lead to catastrophic results, and therefore deadlines are critical.

Soft: results are still useful after the deadline has been missed but not as useful as before the deadline, For example, a stock analysis program. The longer it takes to calculate the results, the price of the stock may rise or decrease and cause the investor to loose money.

In trying to implement any type of real-time application, it is important to understand the deadlines, whether the application can continue if the deadline is not met, and what the results mean to the user if the deadline is not met.

Chapter 3

Soft Real-time OLAP

3.1 Introduction

To achieve soft real-time OLAP, updates from the data warehouse must be merged with the data cube *without taking the system offline or decreasing query resolution times*.

This chapter describes a modifed architecture for the Sidera OLAP server that enables updates to be merged in soft real-time. Our architecture is designed to run on a single machine and exploits the multi-core architecture of newer systems for parallel processing. Our architecture is based on the design of the original Sidera OLAP server, but it has been adapted to run on a single machine, transitioning from a *distributed* architecture to a *shared-memory* architecture. We will explain in Chapter 5 how our shared-memory version of Sidera could be incorporated with the original architecture to add multiple levels of parallelism to Sidera. Our modifications were done to the Sidera indexer and query engine.

We begin with a brief overview of the original Sidera architecture that uses Hilbert Tuple Differential Coding (HTDC) compression. This is followed by a description of

a modified version of the indexer that partitions materialized views into local partitions and uses the "hot" partition mechanism to merge recent updates without taking the system offline or compromising system performance. The modified architecture also exploits multi-core processing, where possible, to increase performance and distribute processing loads across the available processor cores. The new architecture also replaces the older file based buffering subsystem with a faster in-memory based buffering subsystem, and introduces controlled I/O to ensure that files are read or written sequentially to minimize disk seeks.

We end the chapter with a description of the original Sidera query engine followed by the modified architecture. Distributing data across multiple partitions, as well as using a "hot" partition for merging updates puts an extra burden on the query engine. This is because updates are merged only with the tuples in the "hot" partition, which leads to potential duplicate tuples with respect to tuples that are in partitions other than the "hot" partition. For this reason, the query engine must be able to aggregate duplicate tuples on the fly without degrading query resolution times.

3.2 Sidera Overview

Sidera is a paralllel OLAP server that runs on a commodity cluster running the Linux operationg system. It is based on a distributed parallel architecture, where each node functions independently from the others. A Parallel Service Interface (PSI) layer is responsible for coordinating the activities between the nodes. The PSI is build using the standard Message Passing Interface (MPI) API, and abstracts much of the complexity in coordinating communcation between the nodes. Figure 3.1 from [29] shows an overview of the Sidera architecture.

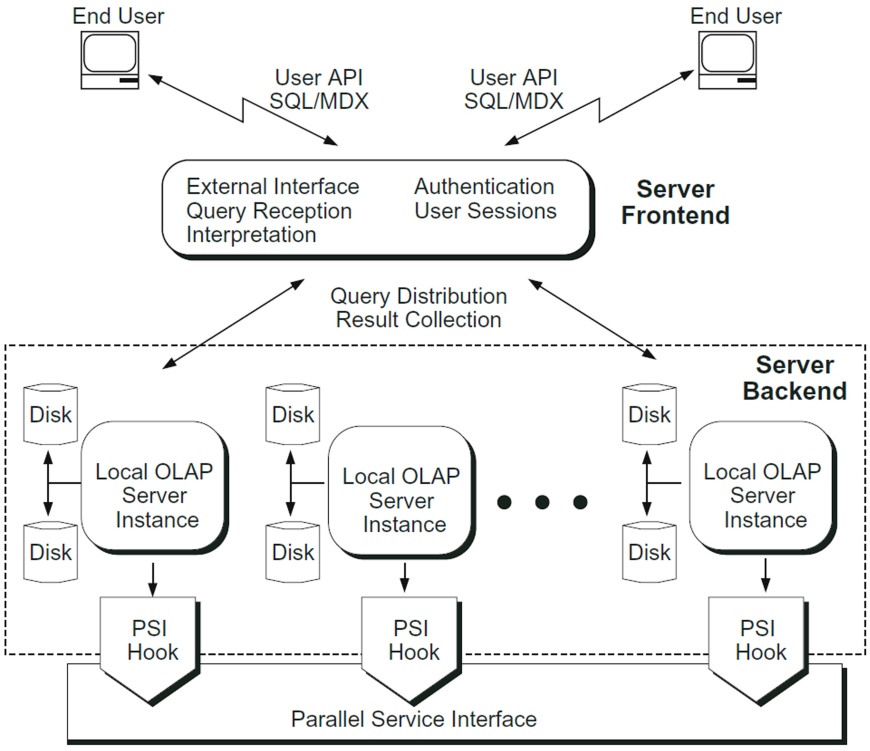


Figure 3.1: The Sidera Architecture.

Each node in the cluster participates in sorting, compression, partitioning, distribution, indexing, and query resolution. Although one of the node is designated as the frontend node, its role is restricted to receiving queries from the user and forwarding them to the backend nodes. This eliminates bottlenecks that are often associated with parallel architectures that rely on a central node for process management.

Sidera uses a Hilbert space filling curve to partition data cubes and distributes them across the cluster. The Hilbert space filling curve has been shown to provide better locality-preservation in discrete multi-dimensional spaces than other space filling curves such as z-order or Gray code. For this reason, it is ideally suited for achieving optimal load balancing when partitioning data cubes across multiple nodes.

Indexes are based on the packed LBF Hilbert R-tree data structure, which are better suited for multi-dimentional indexing than data structures such as bitmaps or B+trees, which are traditionally used in relational databases.

Data cubes as well as indexes are compressed using Hilbert Tuple Differential Coding (HTDC) compression that achieves almost 90% compression ratio [27].

Sidera also supports hierarchical representation, multi-dimensional query caching and approximate query answering.

Sidera follows a series of basic steps to transform and distribute the data cube across the cluster, and then to contruct the HTDC compressed views and indexes. Each step is carried out in parallel, with every backend node participating and using the PSI layer to coordinate the process.

Query resolution works with the same principle. Each backend node holds a partition of any given view, and participates in locating tuples that may match the query parameters. Using the PSI layer to coordinate the process, the backend nodes retrieve matching tuples and work together to merge their local results before returning the final result set to the frontend node.

3.3 The Indexer

The original Sidera indexer algorithm is divided into three stages. In the first stage, each backend node prepares a materialized view by converting the tuples into their corresponding Hilbert indexes and then sorting the view in ascending Hilbert order. In the second stage, the backend node stripes the view to create n partitions, where n is the number of backend nodes in the cluster. It then proceeds to send a partition

to every node including itself, and at the same time receives partitions of other materialized views from the remaining backend nodes. In the third stage, each backend node creates a local HTDC compressed view using the partition it has received in stage two, and builds a compressed Hilbert R-tree index for that view. Stages one to three are repeated until every materialized view has been partitioned and distributed across the cluster.

Stages one and three do not require the PSI layer and run in parallel on each backend node, but stage two depends on the PSI layer to synchronize the distribution of partitions between the nodes. We will now use algorithm 1 to explain the three stages more in detail.

Lines 1 to 9 carry out setup work such as declaring certain variables required by the algorithm, as well as gathering some information about the cluster. We also use the PSI layer to determine which node in the cluster has the highest number of materialized views. This will determine how many times we will cycle through the main part of the algorithm.

Stage one starts at line 10 and runs until line 20. Here, variables required for coordinating the distribution are declared and assigned initial values. If this node has a materialized view to distribute in this round, the tuples are converted to Hilbert indexes, sorted in ascending order and partitioned. Although the calls to ConvertToHilbertIndex($V[i]$) and sortHilbertAscending($V_H[i]$) appear to be both one step processes, they are built on the file based buffering subsystem used in the original Sidera architecture. We will elaborate more on the buffering subsystem in the next section, and explain how we replaced it with an in-memory subsystem to accelerate processing time.

Algorithm 1 Sidera Indexer

Require: Minimal set of materialized views distributed across the cluster.
Ensure: Each backend node has a partition of every view in the minimal set.

1: $V \leftarrow$ minimal set of materialized views
2: $V_H \leftarrow$ minimal set of material views in Hilbert format. ▷ empty at this point
3: $viewCount \leftarrow$ SIZE(V)
4: $viewCountList \leftarrow$ empty list
5: $maxViewCount \leftarrow 0$
6: $n \leftarrow$ number of backend nodes in cluster
7: $id \leftarrow$ id for this node in the cluster
8: ALLGATHER($viewCountList$,$viewCount$,n) ▷ through PSI layer get the maxViewCount from every node into viewCountList
9: $maxViewCount \leftarrow$ MAX($viewCountList$)
10: **for** $i = 1 \rightarrow maxViewCount$ **do**
11: $toReceive \leftarrow$ `false`
12: $toSend \leftarrow$ `false`
13: $sendTo \leftarrow 1$
14: $receiveFrom \leftarrow 1$
15: **if** $i \leq viewCount$ **then** ▷ check if this node has something to send
16: $V_H[i] \leftarrow$ CONVERTTOHILBERTINDEX($V[i]$)
17: SORTHILBERTASCENDING($V_H[i]$)
18: $V_P \leftarrow$ GETPARTITIONS($V_H[i]$)
19: $toSend \leftarrow$ `true`
20: **end if**
21: **for** $j = 1 \rightarrow n$ **do**
22: SEND($toSend$,$sendTo$) ▷ through PSI layer
23: RECEIVE($toReceive$, $receiveFrom$) ▷ through PSI layer
24: **if** $toSend ==$ `true` **then**
25: SENDPARTITION($V_P[j]$, $sendTo$) ▷ through PSI layer
26: **end if**
27: **if** $toReceive ==$ `true` **then**
28: $buffer \leftarrow$ empty buffer to receive partition
29: $buffer \leftarrow$ RECEIVEPARTITION($receiveFrom$) ▷ through PSI layer
30: CREATECOMPRESSEDVIEW($buffer$)
31: BUILDCOMPRESSEDINDEX($buffer$)
32: **end if** ▷ continued on the next page...

Algorithm 2 Sidera Indexer (continued)

```
33:         sentTo ← (sendTo + 1)  mod n
34:         receiveFrom ← (receiveFrom − 1)
35:         if receiveFrom < 1 then
36:             receiveFrom ← n
37:         end if
38:     end for
39: end for
```

Starting at line 21 until line 38, the partitions are distributed across the cluster. Although stage two is the distribution stage, stage three is overlapped with stage two with the compression and index building steps taking place on lines 30 and 31.

Although every backend node repeats lines 10 to 39 until all the materialized views have been distributed, not every node participates in sending and receiving. If a node has less materialized views to send as the $maxViewCount$, then it does not send anything when it has processed all its views. Similarly, if a node is informed on line 23 that a node will not receive a partition, it simply loops to the next round.

On line 18, partitioning is achieved by striping the view such that node j will receive tuples $(j,\ j+n,\ j+2n,\ ...)$ where j is the node id and n is the number of nodes in the cluster. Because of the locality preservation property of the Hilbert space filling curve, tuples that satisfy the parameters of a range query along any dimension usually have corresponding Hilbert indexes that are close together on the Hilbert curve. This means that they are close together when Hilbert indexes are sorted in order. This Hilbert striping strategy therefore ensures that for any given range query, there is balanced distribution of work across the nodes in the cluster.

Algorithm 3 describes how updates are merged with the existing data cube. The process is similar to Algorithm 1. A minimal set of materialized views are generated

Algorithm 3 Sidera Indexer - **Updates**

Require: Minimal set of **update** views distributed across the cluster.

Ensure: Each backend node has merged **updates** with compressed views and rebuilt compressed R-tree indexes.

1: $V_U \leftarrow$ set of **update** views
2: $V_H \leftarrow$ set of views in Hilbert format. ▷ empty at this point
3: $viewCount \leftarrow$ SIZE(V_U)
4: $viewCountList \leftarrow$ empty list
5: $maxViewCount \leftarrow 0$
6: $n \leftarrow$ number of backend nodes in cluster
7: $id \leftarrow$ id for this node in the cluster
8: ALLGATHER($viewCountList$,$viewCount$,n) ▷ through PSI layer get the maxViewCount from every node into viewCountList
9: $maxViewCount \leftarrow$ MAX($viewCountList$)
10: **for** $i = 1 \rightarrow maxViewCount$ **do**
11: $toReceive \leftarrow$ `false`
12: $toSend \leftarrow$ `false`
13: $sendTo \leftarrow 1$
14: $receiveFrom \leftarrow 1$
15: **if** $i \leq viewCount$ **then** ▷ check if this node has something to send
16: $V_H[i] \leftarrow$ CONVERTTOHILBERTINDEX($V_U[i]$)
17: SORTHILBERTASCENDING($V_H[i]$)
18: $V_P \leftarrow$ GETPARTITIONS($V_H[i]$)
19: $toSend \leftarrow$ `true`
20: **end if**
21: **for** $j = 1 \rightarrow n$ **do**
22: SEND($toSend$,$sendTo$) ▷ through PSI layer
23: RECEIVE($toReceive$, $receiveFrom$) ▷ through PSI layer
24: **if** $toSend ==$ `true` **then**
25: SENDPARTITION($V_P[j]$, $sendTo$) ▷ through PSI layer
26: **end if**
27: **if** $toReceive ==$ `true` **then**
28: $buffer \leftarrow$ empty buffer to receive partition
29: $buffer \leftarrow$ RECEIVEPARTITION($receiveFrom$) ▷ through PSI layer ▷

continued on the next page...

Algorithm 4 Sidera Indexer - **Updates** (continued)

```
30:             viewName ← getViewName(buffer)
31:             viewBuffer ← loadCompressedView(viewName)   ▷ decompresses and loads stored view
32:             mergeBuffer(viewBuffer,buffer)   ▷ merges updates in buffer into viewBuffer
33:             createCompressedView(viewBuffer)
34:             buildCompressedIndex(viewBuffer)
35:         end if
36:         sentTo ← (sendTo + 1) mod n
37:         receiveFrom ← (receiveFrom − 1)
38:         if receiveFrom < 1 then
39:             receiveFrom ← n
40:         end if
41:     end for
42: end for
```

that contain the aggregate values for the tuples in the update set, and these views are partitioned and distributed across the cluster. For each update view partition a backend node receives on line 29, the existing compressed view corresponding to that update view must be loaded, decompressed and the updates merged with it. This is reflected with three extra steps from line 30 to 32. Once the updates have been merges, the view is compressed again and the index is rebuilt. The important and time consuming property of the update algorithm is that the *entire* existing view must be loaded and decompressed, and the update tuples must be merged with it. Although materialized views are partitioned across the cluster, these partitions will reach a significant size over time and system performance will decrease when merging updates.

Eavis and Cueva [28] identify Hilbert sorting as the dominant cost in the construction of the LBF R-tree, and since the size of the update set is much smaller than the size of the compressed view, $|U| \ll |r_{data}|$, the cost of merging updates is much smaller

than building the entire data cube. Their update strategy follows the one proposed by Roussopoulos in [61]. In practice, this is not the case because in order to rebuild the compressed index (line 34 of the Algorithm 3), we require the multi-dimensional form of each tuple. This requires every tuples from the compressed view to be converted from their Hilbert index to multi-dimensional format, and this is a computationally expensive operation. In fact, all conversions between multi-dimensional tuples and their respective Hilbert indexes are expensive, and we describe later in this chapter how we leverage multi-core processing to reduce the processing time.

3.4 Indexer Buffering Subsystem

The original Sidera architecture uses a file based buffering subsystem, which allows it to stream data from large views on older systems that have restricted Random Access Memory (RAM). The process of converting a non compressed view to the final HTDC compressed view goes through several intermediate transformations. For each transformation, an intermediate file is generated, which becomes the input for the next transformation. Figure 3.2 shows the four major transformations that the file based buffering subsystem handles.

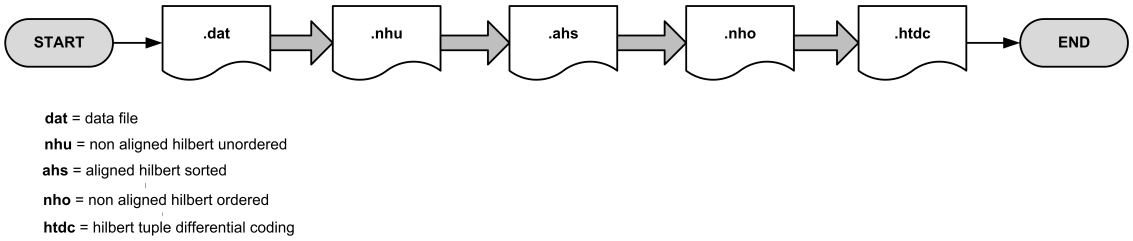


Figure 3.2: Buffering Subsystem - Intermediate Files

The first transformation consists of converting tuples with multi-dimensional coordinates in data file (.dat) into the non-aligned Hilbert unligned format (.nhu). The

(.nhu) file contains the tuples as Hilbert indexes, but they are not yet sorted. Next, we transform the tuples in (.nhu) format into aligned Hilbert sorted (.ahs) format. In this file, the tuples are sorted in ascending Hilbert order, and the Hilbert indexes are stored in a fixed byte format. The original Sidera system uses the GNU Multi-Precision (MP) library to handle and store Hilbert indexes. For high dimension spaces, primitive data types such as *long* may be too small to hold certain Hilbert indexes. Let d be the number of dimensions in a hyper-space, and S is the largest cardinality of any of the axis, then the length of the Hilbert space filling curve is defined as S^d. Essentially, this means that the size of the largest Hilbert index generated by the transformation is S^d. If we take a view with six dimensions, and the cardinality of the largest dimension is 5000. Then the highest possible Hilbert index that can be generate is 5000^6, and this number cannot be stored in a *long* datatype.

The GNU MP library allows integers of arbitrary size to be stored and manipulated. It does this by allocating as many bytes as required to store an integer. This also means that Hilbert indexes stored in GNU MP data types may occupy different number of bytes in memory. The problem arises when the indexes are distributed across the cluster. The PSI layer is built on top of the Message Passing Interface (MPI) API, and requires data types in an array to be of fixed size. For this reason, the Hilbert indexes stored in GNU MP data types must first be transformed into fixed size data types. This is what the transformation from (.nhu) to (.ahs) does.

The transformation from (.ahs) to non-aligned Hilbert ordered (.nho) ordered converts the fixed size data types back to GNU MP data types. The only difference between the (.nhu) and the (.nho) is that the Hilbert indexes in the latter are sorted in ascending Hilbert order. Finally, the last transformation is from (.nho) to Hilbert

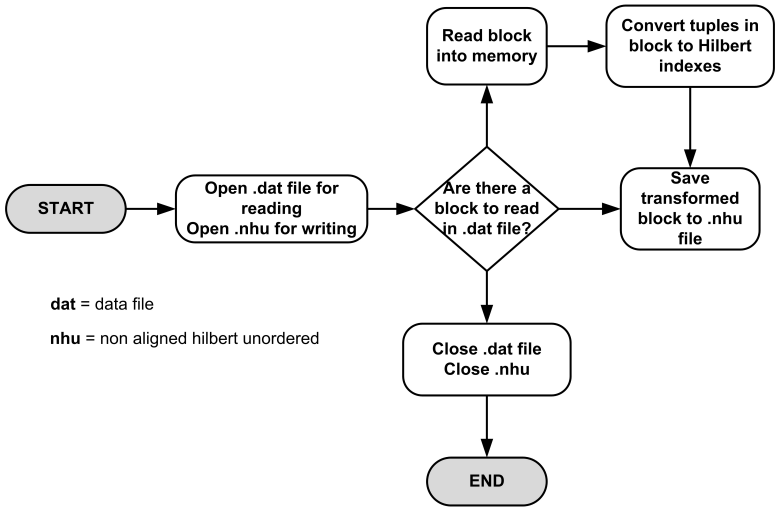


Figure 3.3: File Based Buffering Subsystem

tuple differential coding (.htdc) format, which is the format used by Sidera to store compressed materialized views. After the transformation from (.dat) format to (.htdc) has completed, the intermediate files are delete by the system.

Another way that the original file based buffering subsystem contributes to the Sidera indexer latency is by streaming data one block at a time. Figure 3.3 shows how tuples from the (.dat) file are transformed and stored stored in the (.nhu) format file.

A block is read from the first format, transformed and then written to the next format. This cycle is repeated until every block from the first format has been processed.

Although allowing older systems to process larger files sizes, the file based buffering subsystem also has a negative impact on the run-time. It forces the application to execute a large number of disk operations because (a) each block is individually read and written, requiring two I/O calls for each block, (b) it must create a series of

intermediate files and (c) reading from one file and directly writing to another file forces the disk to perform seek operations. These all contribute to increase the run-time of the original Sidera server.

In our modified architecture, we have replaced the file based buffering subsystem with a bi-directional in-memory subsystem. Figure 3.4 provides an abstract view of how this subsystem functions. For each materialized view, the view is first read sequentially from the (.dat) format into memory and all transformations take place in memory. At the end of the last transformation, the compressed view in (.htdc) format is written sequentially onto disk. All read and write calls are issued as a single call.

This approach reduces the number of I/O calls to a minimum and eliminates the disk seek operations incurred when overlapping I/O operations were issued for two separate files. Our subsystem can do transformations in both directions, making it also suitable for the Sidera query engine, which is explained a little further in this chapter.

3.5 Indexer with Local Partitioning

We now present a single process indexer for Sidera that locally partitions materialized views. Based on the original architecture of the Sidera indexer, it stores partitions on the same machine instead of distributing them across the cluster. We explain a little further in this chapter how we leverage multi-core processing to achieve parallel processing on a single machine. Effectively, our modifications transform Sidera from a *distributed* architecture to a *shared-memory* architecture. We propose in Chapter 5 a design to retro-fit this *shared-memory* version of Sidera into the original *distributed*

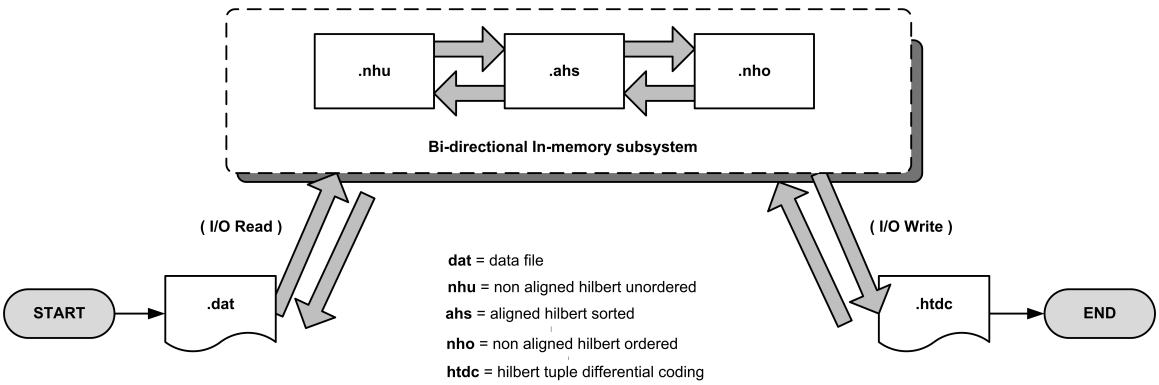


Figure 3.4: Bi-directional In-memory Subsystem

architecture to achieve multi-level parallelism in the Sidera OLAP server. Algorithm 5 presents the core algorithm of the indexer when the data cube is compressed and indexed.

From line 1 to 5, the algorithm prepares the necessary structures for processing. The number of partitions p can be any value such that $p \geq 1$. In the distributed version of Sidera, the number of partitions was dependent on the number of nodes n in the cluster. In the shared-memory version, this is no longer the case.

Line 4 declares a buffer that is the in-memory subsystem described in Section 3.4. All transformation and partitioning tasks are performed by this structure. Another data structure, called the *partition table* is declared on line 5. The partition table stores information about each view, such as number of partitions, maximum partition size and if a "hot" partition exists on disk. At the end of the indexer process on line 15, the partition table is written to disk. This file will be used later by the indexer to

Algorithm 5 Sidera Indexer with Local Partitioning

Require: Minimal set of materialized views on a single machine.

Ensure: Each view has been partitioned, compressed and has an Hilbert R-tree index.

1: $V \leftarrow$ minimal set of materialized views
2: $viewCount \leftarrow$ SIZE(V)
3: $p \leftarrow$ number of partitions to generate
4: $B_I \leftarrow$ in-memory subsystem ▷ The in-memory subsystem takes care of transformations and partitioning once the view is loaded
5: $T_P \leftarrow$ partition table ▷ Holds information about views and partitions
6: **for** $i = 1 \rightarrow viewCount$ **do**
7: LOAD(B_I, $V[i]$)
8: $V_P \leftarrow$ GETPARTITIONS(B_I)
9: **for** $j = 1 \rightarrow p$ **do**
10: CREATECOMPRESSEDVIEW($V_P[j]$)
11: BUILDCOMPRESSEDINDEX($V_P[j]$)
12: UPDATEPARTITIONTABLE(T_P,V_P)
13: **end for**
14: **end for**
15: WRITETODISK(T_P) ▷ write partition table to disk

merge updates and by the query engine to quickly determine the number of partitions to query and if "hot" partitions exists, or whether a new "hot" partition should be created.

From line 6 to line 14, each view is loaded, transformed, compressed and the R-tree index is build. The process is the same for every materialized view.

Finally, we note that I/O read calls are restricted to line 7 where the entire view file is loaded in one read operation, line 10 where the compressed view is written in one operation, line 12 where the compressed index is written in one operation, and line 15, when the partition table is written to disk.

3.6 The Partition Table

As mentioned in the previous section, the partition table is a simple structure that keeps tracks of materialized views, partitions, "hot" partitions, and the maximum size before capping a "hot" partition. The table is initially create and stored when the data cube is first generated, and it is updated by the update process. The query engine also relies on the partition table to determine how many partitions to query and whether it should search for a "hot" partition during query resolution.

3.7 Tuple Distribution on a Shared-memory Architecture

In the distributed memory Sidera architecture, partitioning is achieved by striping the view such that node j will receive tuples $(j,\ j+n,\ j+2n,\ ...)$ where j is the node id and n is the number of nodes in the cluster. This is done to ensure that there is a balanced work distribution during query resolution. We refer to this as *Hilbert*

striping. In a shared-memory architecture, we are more interested in minimizing the number of blocks retrieved during query resolution. Once the blocks are loaded in memory, they can be distributed among the processing cores because each block is compressed at the block level and can be decompressed independently. Hilbert striping would make our architecture produce slower query times by spreading out the tuples in a range query across many blocks. For this reason, our shared-memory architecture compresses Hilbert indexes that are close together in the same block instead of striping them into different blocks. We refer to this strategy as *Hilbert packing*.

3.8 Merging Updates and the"hot" Partition

When merging updates, the modified architecture begins with a set of update views. Each update view is sequentially loaded in the in-memory buffer, and the tuples are converted to Hilbert indexes and sorted in ascending order. The algorithm then tries to locate the compressed "hot" partition corresponding to the update view. If a "hot" partition is found, it is loaded and decompressed. The Hilbert indexes of the update view are merged with those of the "hot" partition, and then they are re-compressed and saved to disk as the "hot" partition. If the number of the tuples in the "hot" partition exceeds the maximum allowed, the "hot" partition is capped with the maximum number of tuples, and the remaining tuples are stored in a new "hot" partition. If the indexer cannot find a "hot" partition, it simply starts a new one with the tuples in the update view. This is a repeated for every update view in the update set. Algorithm 6 shows how updates are merged with the "hot" partition in the Sidera indexer.

Algorithm 6 Sidera Indexer with Local Partitioning - **Updates**

Require: Set of **updates** views on a single machine.
Ensure: Each view has been partitioned, compressed and has an Hilbert R-tree index.

1: $U \leftarrow$ set of update views
2: $viewCount \leftarrow$ SIZE(U)
3: $B_I \leftarrow$ in-memory subsystem ▷ The in-memory subsystem takes care of transformations and merging buffers
4: $T_P \leftarrow$ LOADPARTITIONTABLE ▷ load partition table from disk
5: **for** $i = 1 \rightarrow viewCount$ **do**
6: LOAD(B_I, $U[i]$)
7: $viewName \leftarrow$ GETVIEWNAME($buffer$)
8: $maxTuples \leftarrow$ GETMAXPARTITIONSIZE
9: **if** HOTPARTITIONEXIST($viewName$) == `true` **then**
10: $hotPartBuffer \leftarrow$ LOADHOTPARTITION($viewName$) ▷ decompresses and loads hot partition for this view
11: MERGEBUFFER(B_I,$hotPartBuffer$) ▷ merges tuples in `hotPartBuffer` with **updates** in B_I
12: **end if**
13: $V_P \leftarrow$ GETPARTITIONS(B_I, $maxTuples$) ▷ We cap a partition if it has number of tuples = $maxTuples$
14: **for** $j = 1 \rightarrow p$ **do**
15: **if** $j == p$ & SIZE($V_P[j]$) $< maxTuples$ **then** ▷ last partition not full. We create a compressedhot partition
16: CREATEHOTPARTITION($V_P[j]$)
17: BUILDHOTPARTITIONINDEX($V_P[j]$)
18: **else**
19: CREATECOMPRESSEDVIEW($V_P[j]$)
20: BUILDCOMPRESSEDINDEX($V_P[j]$)
21: **end if**
22: UPDATEPARTITIONTABLE(T_P,V_P)
23: **end for**
24: **end for**
25: WRITETODISK(T_P) ▷ write partition table with updates to disk

Lines 1 to 4 execute set up tasks, which includes loading the partition table. The partition table holds information about existing materialized views, how many partitions per view and what is the maximum number of tuples the "hot" partition of any view can hold. From line 5 to line 24, the update views are merged with the "hot" partition. Lines 6 to 8 load the update view, and partition information on that view is obtained from the partition table. If a "hot" partition already exits, lines 10 to 11 execute and the "hot" partition is loaded and merged with the tuples of the update view.

We then retrieve on line 13 a set of partitions from the in-memory buffer. It automatically determines how many capped partitions should be created and how many remaining tuples should be pushed into the "hot" partition. The HTDC compression and index building for each partition is done from line 15 to line 23. Note that if the last partition in the partition set has less tuples than the maximum number of tuples retrieved on line 8, then the partition is marked as a "hot" partition. Otherwise, all partitions are created as capped partitions. Finally, the updated partition table is stored to disk on line 25, and it is ready for the next round of updates.

The update algorithm for the modified architecture is very close to the update algorithm of the original architecture, except that we now merge with the "hot" partition instead of the entire view. By restricting the size of the "hot" partition to a maximum number of tuples, we enforce an upper bound on the merge time. In the original Sidera indexer, the stored view becomes larger with time as updates are merged, and the merge time increases linearly as the size of the view increases. A negative effect of capping partitions as is done in our architecture is that the partitions created after the initial data cube was built may contain duplicate tuples since the

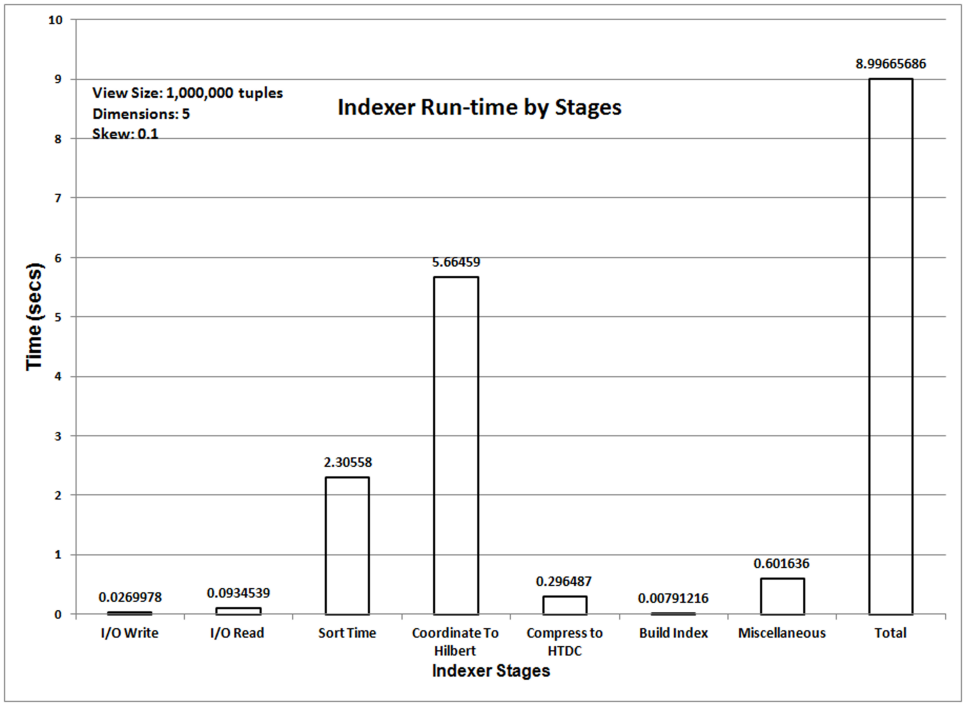


Figure 3.5: Run-time by Stages for Indexer with Partitioning

merge process only merges with the "hot" partition. For this reason, the modified query engine presented a little later in this chapter has to aggregrate tuples on the fly, which increases the query resolution time. We also explain later how we use multi-core processing in the modified query engine to mitigate this.

3.9 Indexer with Multi-core Processing

Before adding multi-core processing to the Sidera indexer, we first profiled the run-time of our indexer with local partitioning that was described in Section 3.5. Figure 3.5 shows a distributation of the run-time for the major stages of the indexer.

The run-time was measured by using a single materialized view with 5 dimensions, 1,000,000 tuples, and a skew factor of 0.1.

The profile shows that run-time is dominated by the *Coordinate To Hilbert* and the *Sort Time* stages. Together, they constitute almost 89% of the total run-time. Both of these stages take place in the in-memory buffering subsystem described in Section 3.4. The *Coordinate To Hilbert* stage converts tuples in coordinate format to their respective Hilbert indexes. Each conversion is an independent step, so converting an array of tuples can be done in parallel. The *Sort Time* stage sorts the Hilbert indexes generated in *Coordinate To Hilbert* in ascending order. This is done using a basic qsort algorithm. To take advantage of multi-core processing, we apply a parallel sample sort algorithm that was adapted for a shared-memory architecture.

We examine the next important stages with respect to run-time. The *Miscelleneous* encompasses all run-time related to initializing the GNU MP variables used to hold Hilbert indexes, creating partitions, calculating cardinalities, and calculating the Hilbert differences that are used by the HTDC compression. These are tasks performed by the in-memory buffering subsystem. Of these, the GNU MP initialization can be parallelized because each initialization is independent from the other.

The *Compress to HTDC* cannot be parallelized because the process does not know beforehand how many tuples in Hilbert index format can be compressed in a block until the compressed block is created. So sequentially compressed blocks must wait until the block before them is ready before proceeding.

The *I/O Read* and the *I/O Write* stages are difficult to parallelize on a single disk platform. We have already minimized the I/O by using the in-memory buffering structure, and performing read and write operations for entire files instead of one block at a time. This minimizes the number of I/O calls and the number of disk seeks. Performing parallel operations would require specialized hardware such as a

disk array with parallel I/O support.

Finally, the *Build Index* stage has a negligeable effect on the run-time, and the overhead of parallelizing this stage would far outweigh any benefits gained from it.

We now present the algorithms implemented within the in-memory buffering subsystem that execute the coordinate to Hilbert conversion as well as the Hilbert ascending sort with multi-core processing.

HIlbert conversions are implemented using Hilbert transformation libraries developed by Moore [55]. Since each conversion is an independent operation, we extended Doug Moore's libraries to be thread safe, and implemented a parallel algorithm to perform the conversions. Algorithm 7 shows the transformation from multi-dimensional tuples to Hilbert indexes with multi-core processing. The main thread executes lines 1 to 4. At line 5, the main thread forks and spawns the threads that will run in parallel. Lines 6 to 11 are executed by each thread, where each thread transforms $\frac{n}{coreCount}$ tuples, where n is the number of tuples in a view, and $coreCount$ is the number of threads used in the parallel section.

Multi-core processing starts at line 5 and ends at line 12. The OpenMP library handles the fork and join operations, as well as work distribution. The number of threads used for multi-core processing section may be the same as the number of physical cores, but this is not necessary. It can be any number specified by the process.

In Algorithm 8, we present a parallel sample sort algorithm. The array of unsorted Hilbert indexes H is partitioned into n partitions, where n is the number of threads. Each thread uses qsort to sort its partition in ascending order. Once this is done, each thread select $n - 1$ indexes from its sorted partition, and these are brought together

Algorithm 7 Convert Tuples to Hilbert Indexes - Multi-core Processing

Require: A set of tuples in coordinate format.
Ensure: A set of corresponding Hilbert indexes.

```
1: V ← a view with tuples
2: H ← empty set of Hilbert indexes
3: d ← number of coordinate tuples in V
4: coreCount ← number of threads used
5:                                          ▷ Start multi-core processing here
6: q ← core id
7: for i = 1 → d do
8:     if i mod coreCount == q then
9:         convert tuple V[i] to Hilbert index and set into H[i]
10:    end if
11: end for
12:                                         ▷ End multi-core processing here
```

in an array of size $n(n-1)$. This array is sorted, and $n-1$ indexes are again selected at regular intervals. These values act like the "pivot" value in qsort.

Next, each threads scans it sorted partition. Values that are less than S_1, where S is the array containing $n-1$ indexes, are sent to core 1. Values that are less than S_2 but greater than or equal to S_1 are sent to core 2, and so on. Since each core scans through a sorted partition, the lists received by each core are also sorted and only require merging.

At the end of the step, each core has a sorted partition of Hilbert indexes, such that all values on core 1 are less than S_1, all values on core 2 are less than S_2 but greater than or equal to S_1 and so on. The array H is now sorted in ascending order.

3.10 Query Engine

The original Sidery query engine algorithm has two phases. The first phase starts when a backend node receives the query parameters from the frontend. The Linear

Algorithm 8 Parallel Sample Sort with Multi-core Processing

Require: An array of unsorted Hilbert indexes.
Ensure: An array of sorted Hilbert indexes.

Start Sequential Processing
$H \leftarrow$ array of unsorted Hilbert indexes
$numIndexes \leftarrow$ SIZE(H)
$coreCount \leftarrow$ number of processing cores
$sortSize \leftarrow numIndexes \div coreCount$
Start Parallel Processing on n **cores**
$coreId \leftarrow$ core id of this core
$startIndex_{coreId} \leftarrow (coreId$ - 1) $\times$ $sortSize$
$endIndex_{coreId} \leftarrow coreId \times sortSize$
QSORT(H, $startIndex$, $endIndex$)
$S_{coreId} \leftarrow$ ($coreCount$ - 1) values from H between $startIndex$ and $endIndex$ at regular intervals
$B \leftarrow$ empty list buffer
for $i = 1 \rightarrow coreCount$ **do**
    $B \leftarrow$ append(S_i)
end for
QSORT(B)
$S_g \leftarrow$ ($coreCount$ - 1) values from B at regular intervals
$P \leftarrow$ array of size $coreCount$ of lists
for $i = 1 \rightarrow sortSize$ **do**
    $T \leftarrow$ getTupleAt(H,i + startIndex)
    for $j = (p - 1) \rightarrow 1$ **do downto**
        if $T > S_g[j]$ **then**
            pushback(P_j, T)
            break inner for loop
        end if
    end for
end for
for $i = 1 \rightarrow coreCount$ **do**
    $recordCount \leftarrow$ size(R_i, i)
    for $j = 1 \rightarrow recordCount$ **do**
        $T \leftarrow R_i[j]$
        pushback(R_g,T)
    end for
end for
RETURN R_g

Breath First Search (LBFS) algorithm traverses the R-tree index. Every time a leaf node is reached, the corresponding data block is located and loaded from the HTDC compressed view file. The block is then decompressed and the tuples in the block are checked sequentially to see if any satisfy the query parameters. Valid tuples are added to the result set. This process continues until the LBFS algorithm has completed traversing the index. Algorithm 9 presents phase 1 of the original query engine.

Phase 1 is carried out independently on each backend node without requiring the PSI layer. At the end of the phase 1, each backend node has a local result set that is ready to be merged with the result sets of the other nodes. A careful examination of algorithm 9 shows that both the compressed index file as well as the compressed view file are read from file one block at a time. For large files, especially compressed view files, this translates to a large number of I/O calls. A second characteristic of this algorithm is that I/O calls to read from the index file are overlapped with the I/O calls to read from the view file. Alternatively reading from the index file and view file forces the disk to seek constantly. Furthermore, both the DECOMPRESSINDEXBLOCK and DECOMPRESSDATABLOCK procedures take place in memory and each compressed block can be decompressed independently from the other blocks.

In the second phase of the original query engine, each backend node starts with a local result set, and using the PSI layer, the backend nodes sort and merge the local result sets, aggregating duplicate values when necessary, and return an aggregated global result set.

Using a parallel sample sort algorithm, each backend node first sorts its local result set using the qsort algorithm and then creates an array S with $(p-1)$ sample

Algorithm 9 Query Engine - Phase 1

Require: User query, corresponding HTDC compressed view and R-tree index.
Ensure: Result set with tuples that correspond to parameters in user query.

```
1: F_d ← HTDC compressed view corresponding to query parameters
2: F_i ← HTDC compressed R-tree index corresponding to query parameters
3: Q ← empty queue
4: B ← readBlock(F_i)
5: decompressIndexBlock(B)
6: R ← empty result set
7: enqueue(B,Q)
8: while Q is not empty do
9:     B ← dequeue(Q)
10:    if isLeafNode(B) then
11:        OFFSET ← getOffset(B)
12:        seek(F_d,OFFSET)
13:        D ← readBlock(F_d)
14:        decompressDataBlock(D)
15:        tuples ← getTupleCount(D)
16:        for i = 1 → tuples do
17:            T ← getTuple(D,i)
18:            if T matches query parameters then
19:                add(T,R)
20:            end if
21:        end for
22:    else
23:        children ← getChildCount(B)
24:        for i = 1 → children do
25:            child ← getChildEntry(B,i)
26:            if child bounding satisfies query parameters then
27:                OFFSET ← getOffset(child)
28:                seek(F_i,OFFSET)
29:                C ← readBlock(F_i)
30:                decompressIndexBlock(C)
31:                enqueue(C,Q)
32:            end if
33:        end for
34:    end if
35: end while
36: RETURN R
```

Algorithm 10 Query Engine - Phase 2 (Parallel Sample Sort)

Require: p unsorted recordsets R where p is the backend node count.
Ensure: merged and sorted global recordset R_G

1: $R_p \leftarrow$ unsorted local recordset on node p
2: $R_g \leftarrow$ empty global recordset
3: qsort(R_p)
4: $S_i \leftarrow$ (p - 1) values from R_p at regular intervals
5: $B \leftarrow$ empty list buffer
6: **for** $i = 1 \rightarrow p$ **do**
7: $B \leftarrow$ append(S_i)
8: **end for**
9: qsort(B)
10: $S_g \leftarrow$ (p - 1) values from B at regular intervals
11: $P \leftarrow$ array of size p of lists
12: $recordCount \leftarrow$ size(R_p)
13: **for** $i = 1 \rightarrow recordCount$ **do**
14: $T \leftarrow$ getTuple(R_p,i)
15: **for** $j = (p - 1) \rightarrow 1$ **do**
16: **if** $T > S_g[j]$ **then**
17: pushback(P_j, T)
18: break inner for loop
19: **end if**
20: **end for**
21: **end for**
22: **for** $i = 1 \rightarrow p$ **do**
23: $recordCount \leftarrow$ size(R_i, i)
24: **for** $j = 1 \rightarrow recordCount$ **do**
25: $T \leftarrow R_i[j]$
26: pushback(R_g,T)
27: **end for**
28: **end for**
29: RETURN R_g

values where p is the number of nodes in the cluster. The values selected are exactly p intervals apart. Each node then broadcasts its values to the rest of the cluster using the PSI layer such that each backend node ends up with a list with the sample values from every node. This list is sorted using qsort and the array S is populated again by selecting $(p-1)$ for the sorted list. In the next step, each node scans through its recordset and identifies every tuple that is less than S_1 and transmits the tuples to node 1, then every tuples less than S_2 but greater than or equal to S_1 to node 2 and so on. The last node receives every tuple that is greater than or equal to S_{p-1}. The list of tuples that are transmitted between nodes are already sorted, so when receiving a list of tuples from another node, a backend node simple merges it with the tuples it already has in its buffer. The final step to phase 2 is assembling the local lists into a single global recordset from node 1 to node p by appending them sequentially using the PSI layer and delivering the results to the frontend node.

3.11 Query with Partition

We now present the modified query engine that handles partitions and uses a in-memory buffering system. The key to the modified query engine is that each major action is separated into distinct steps, which makes it simpler to apply multi-core processing. In this section, we describe the query engine without multi-core processing, and like the Sidera indexer, it has been adapted to run on a single node. The PSI layer has been removed. Instead of the sample sort algorithm, we use a simple qsort and aggregate duplicate tuples. We will re-implement the sample algorithm with multi-core processing in the next section.

We identify the distinct steps as I/O reads, Linear Breath First Search (LBFS),

data decompression and tuple matching, and finally merging result sets. The I/O reads are further devided into reading the index file and reading the view file. The most important is separating the I/O from the other processing steps. We use an in-memory buffer to load the compressed index file. We traverse the index using the LBFS, keeping a list of the leaf nodes that satisfy the query parameters. Once the LBFS algorithm has identified all potential view blocks, we load the blocks in a sequential manner. Once all the blocks are loaded in memory, they are decompressed and searched for matching tuples. This minimizes the number of disk calls and also the number of seeks, both major contributors to long run-times. Algorithm 11 shows the five distinct steps in the modified algorithm. We expand on *process data blocks* and *merge result sets* in algorithms 12 and 13 since these are the two steps that will also be implemented with multi-core processing.

Algorithm 11 Query Engine - Modified

Require: User query, corresponding HTDC compressed view and R-tree index.
Ensure: Merged and Sorted Global Recordset R_G
1: Load Indexes
2: LBF Search
3: Load Data Blocks
4: Process Data Blocks
5: Merge Result Sets

3.12 Query Engine with Multi-core Processing

After modifying the query engine to run on a single backend node and to process partitions, we now turn our attention to multi-core processing. The steps in our modified architecture that we could adapt to multi-core processing were *process data blocks* and *merge result sets*. In algorithm 14, we take advantage of the fact that view blocks are

Algorithm 12 Query Engine - Process Data Blocks

Require: List of matching data blocks loaded in memory
Ensure: Recordset of matching tuples.

```
1: D ← list of matching blocks loaded in memory
2: R ← empty result set
3: blockCount ← size(D)
4: for i = 1 → blockCount do
5:     B ← dequeue(D)
6:     decompressDataBlock(B)
7:     tuples ← getTupleCount(D)
8:     for j = 1 → tuples do
9:         T ← getTuple(B,i)
10:        if T matches query parameters then
11:            add(T,R)
12:        end if
13:    end for
14: end for
15: return R
```

Algorithm 13 Query Engine - Merge Result Sets

Require: Recordset of tuples.
Ensure: Recordset of tuples sorted and duplicates aggregated.

```
1: R ← recordset of tuples.                ▷ from Process Data Blocks procedure
2: R_agg ← empty recordset.
3: qsort(R)
4: recordCount ← size(R)
5: T_prev ← getTuple(R, 1)
6: for i = 2 → recordCount do
7:     T_next ← getTuple(R,i)
8:     if T_prev == T_next then
9:         T_prev.measure+ = T_next.measure
10:    else
11:        pushback(R_agg,T_prev)
12:        T_prev ← T_next
13:    end if
14: end for
15: return R_agg
```

compressed using a block level compression framework (HTDC compression) and we distribute blocks among the threads and let each tread return a local result set. Next in algorithm 15, we implement a sample sort to replace the qsort.

Algorithm 14 Query Engine - Process Data Blocks with Multi-core Processing

Require: List of matching data blocks loaded in memory
Ensure: Local recordset of matching tuples for blocks processed by the thread.

```
1: D ← list of matching blocks loaded in memory.          ▷ from previous steps
2: R_L ← empty local result set
3: blockCount ← size(D)
4: coreCount ← number of cores used to process blocks
5: id ← getThreadID
6: for i = 1 → blockCount do
7:     if i  mod coreCount == id then
8:         B ← dequeue(D)
9:         decompressDataBlock(B)
10:        tuples ← getTupleCount(D)
11:        for j = 1 → tuples do
12:            T ← getTuple(B,i)
13:            if T matches query parameters then
14:                add(T,R_L)
15:            end if
16:        end for
17:    end if
18: end for
19: return R_L
```

Algorithm 15 Query Engine - Merge Result Sets with Multi-core Processing

Require: *coreCount* unsorted recordsets R where *coreCount* is the number of cores used in Algorithm 14

Ensure: merged and sorted global recordset R_G

```
1: p ← core id of this core
2: R_p ← unsorted local recordset for this core
3: R_g ← empty global recordset
4: qsort(R_p)
5: S_i ← (p - 1) values from R_p at regular intervals
6: B ← empty list buffer
7: for i = 1 → p do
8:     B ← append(S_i)
9: end for
10: qsort(B)
11: S_g ← (p - 1) values from B at regular intervals
12: P ← array of size p of lists
13: recordCount ← size(R_p)
14: for i = 1 → recordCount do
15:     T ← getTuple(R_p,i)
16:     for j = (p − 1) → 1 do
17:         if T > S_g[j] then
18:             pushback(P_j, T)
19:             break inner for loop
20:         end if
21:     end for
22: end for
23: for i = 1 → p do
24:     recordCount ← size(R_i, i)
25:     for j = 1 → recordCount do
26:         T ← R_i[j]
27:         pushback(R_g,T)
28:     end for
29: end for
30: RETURN R_g
```

Chapter 4

Results

4.1 Hardware and Software Specifications

The experiments were run on a custom system comprised of a single AMD Opteron® 6172 processor with a cpu core frequency of 2100 MHz. This processor has a total of 12 physical cores, a dedicated 128K L1 cache for each core, a dedicated 512K L2 cache for each core, and a 12MB L3 cache is also shared between the cores. The server is also equiped with 24 GiB of RAM, and 8 Serial ATA hard drives of 1 TiB in RAID 5 configuration. The server is part of a heterogenous 20 server cluster called the Sidera cluster. The test machine is designated as the front end node of the cluster.

The software components relevant to our experiments:

- CentOS Linux Release 6.0 using kernel 2.6.32.71.el6.x86_64, and GNOME 2.28.2 for the UI
- gcc/g++ version 4.4.4.2010726
- stdlibc++ version
- OpenMP v3.0 for x86_ 64 architecture

The development tools relevant to our experiments:

- Eclipse IDE for Parallel Application Developer version Indigo Service Release 1
- PTP Parallel Tools Platform version 5.04.201111121445
- Subsclipse 1.6.18
- Eclipse CDT 8.0.0.201109151620
- NOMACHINE NX server for Linux version 3.2.0-74-SVN OS
- NOMACHINE NX client for Windows Version 3.4.0-7

The data sets and queries used for the experiments were generated using custom generator tools. The data set generator tool allowed us to specify the number of dimensions, the cardinality for each dimension, the number of tuples to generate, and a zipfian skew between 0 and 1. The skew ensures that the generated data is not simply randomly distributed, but rather follows a more "real life" scenario, which means that there exist clusters of tuples in the data sets. For each experiment, a collection of 10 data sets and update sets were generated, and the run-time was determined by calculating the average of the collection. Similarly, for each query engine experiment, a collection of 10 queries were generated, and the run-time was determined by calculating the average of the collection.

4.2 Building and Updating the Datacube and Indexes

The first set of experiments are designed to test the Sidera indexer. Specifically, we measure the time it takes to build an HTDC compressed data cube and indexes, as

well as updating the data cube with an update set. The tests include running the indexer without partitioning and multi-core to create a baseline measurement with a series of generated data sets, and then running the indexer with partitioning and multi-core processing enabled against the same data sets. The goal is to evaluate the performance of the indexer with partitioning and multi-core processing with respect to the baseline measurements.

4.2.1 Varying Number of Records

In this test, we generated three data sets with 6 dimensions, and a skew of 0.1. The three data sets consists of a 100,000, 1,000,000 and 10,000,000 tuples respectively. The number of threads used for multi-core processing was set to 12, the number of initial partitions created was 4 when partitioning was used. The maximum size of the "hot" partition was 21000 tuples. The update sets were 1% the size of the data sets, i.e. when using the 1,000,000 tuples data set, we generated an update set with 10,000 tuples.

	Run-times in Seconds		
Tuples	No Partitioning	Partitioning	Part. with Multi-core
100,000	0.8304	0.8375	0.3278
1,000,000	8.2569	9.3445	2.3094
10,000,000	87.6562	101.6653	20.6765

Table 4.1: Indexer Build Run-times using Data Sets of Different Sizes.

The results for building the data cube and indexes are shown in Table 4.1. The indexer with partitioning and without multi-core processing is slightly slower than the baseline indexer, which does not use partitioning. Figure 4.1(a) shows that the performance difference increases as the data set gets largers. When using a data set with 10,000,000 tuples, we see an almost 16% increase in run-time.

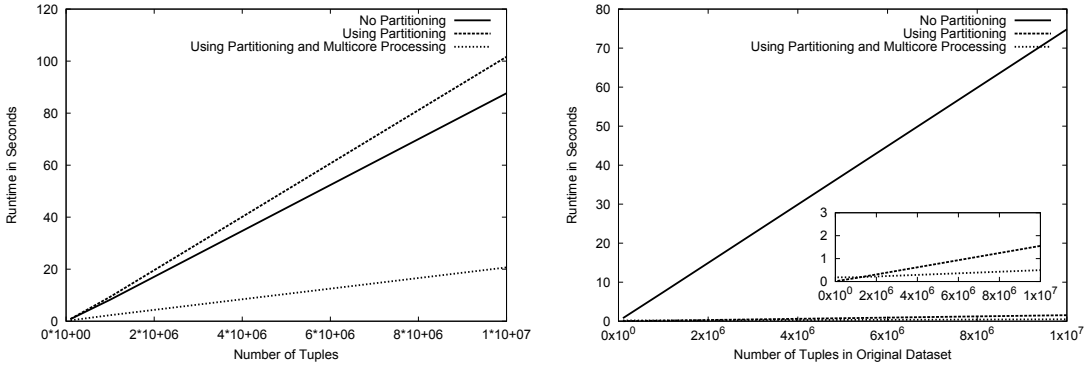
Tuples	Update Set	Run-times in Seconds		
		No Partitioning	Partitioning	Part. with Multi-core
100,000	1,000	0.7896	0.02649	0.1812
1,000,000	10,000	7.4658	0.1538	0.1865
10,000,000	100,000	74.8119	1.5534	0.4983

Table 4.2: Indexer Update Run-times using Data Sets of Different Sizes.

The indexer with partitioning and multi-core processing performs better that the baseline. The results show a performance increase of over 76% when using a data set with 10,000,000 tuples. Figure 4.1(a) also indicate that as the data sets become larger, the indexer with partitioning and multi-core processing offers better speedup with respect to the baseline.

The results for updating the data cube and indexes are presented in Table 4.2. When pushing updates into the data cube, both indexers with partitioning perform better than the baseline. When the original recordset and the update set are smaller, the indexer with partitioning and without multi-core processing provides the best performance. As the original data set and the update set becomes larger, the indexer with partitioning and multi-core processing offers the best performance.

With an original data set of 10,000,000 tuples, an update set of 100,000 tuples, the indexer with partitioning shows more than a 97% performance increase over the baseline. Under the same conditions, the indexer with partitioning and multi-core processing has more than a 99% increase when compared to the baseline. Furthermore, Figure 4.1(b) shows that both indexers with partitioning perform better compared than the baseline as the size of the original recordset and update set increases.



(a) Build Process (b) Update Process

Figure 4.1: Indexer Run-times using Recordsets of Varying Number of Tuples.

	Run-times in Seconds		
Dimensions	No Partitioning	Partitioning	Part. with Multi-core
4	8.0751	8.5765	2.0507
6	8.2569	9.3445	2.3095
8	10.2716	10.8595	2.8905

Table 4.3: Indexer Build Run-times using Different Number of Dimensions.

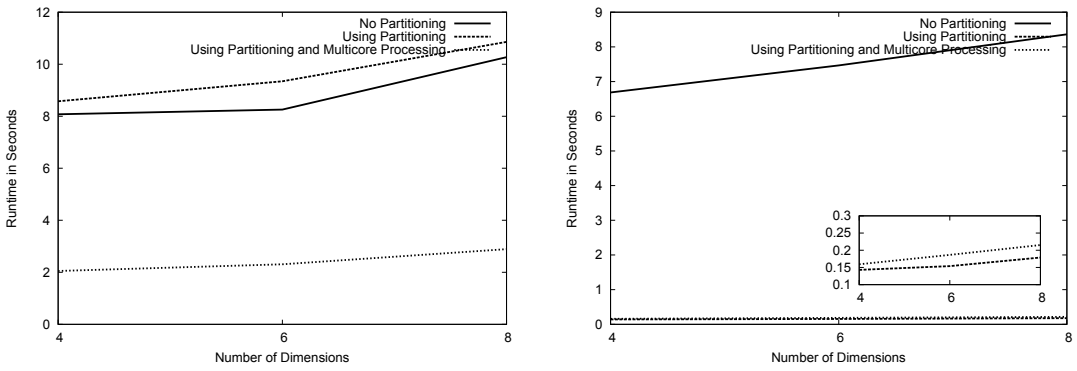
4.2.2 Varying Number of Dimensions

In this test, we generated three data sets with the same specifications as those described in Section 4.2.1, with the difference that we vary the number the number of dimensions instead of the number of tuples. Our data sets have dimensions 4, 6 and 8 respectively. All three recordsets have 1,000,000 tuples.

The results for building the data cube and indexes are given in Table 4.3. For each indexer, the run-time increases slightly as the number of dimensions increases. Figure 4.2(a) shows that the indexer using partitioning and not using multi-core processing does not perform as well as the baseline indexer, but the indexer with both partitioning and multi-core processing has better performance for all three data sets.

	Run-times in Seconds		
Dimensions	No Partitioning	Partitioning	Part. with Multi-core
4	6.6862	0.1430	0.1592
6	7.4658	0.1538	0.1865
8	8.3623	0.1789	0.2153

Table 4.4: Indexer Update Run-times using Recordsets with Varying Dimensions.



(a) Build Process (b) Update Process

Figure 4.2: Indexer Run-times using Recordsets with Varying Dimensions.

When using a data set with 6 dimensions, we see a 13% increase in run-time when using partitioning without multi-core processing compared to the baseline. Applying multi-core processing, the results show over 72% performance increase compared to the baseline.

The results for updating the data cube and indexes are presented in Table 4.4. Using an original data set with 6 dimensions, the indexer using partitioning and not using multi-core processing has a 98% performance increase compared to the baseline indexer. The indexer with partitioning and multi-core processing provided a 97% increase when compared to the baseline. Refering to Figure 4.2(b), we see that both indexers with partitioning demonstrate that using a "hot" partition for merging updates provides performance improvement over the original Sidera indexer.

4.2.3 Varying Number of Threads

For this test, we generated three data sets with 6 dimensions, 1,000,000 tuples, and a skew of 0.1. An update set with 10,000 tuples was also generated. The number of initial partitions was 4 when partitioning was used, and the maximum size of the "hot" partition was 21000 tuples. The number of threads used for multi-core processing was varied between 1 and 48 threads. The number of physical cores available on our test system was 12.

	Run-times in Seconds	
# Threads	No Partitioning	Part. with Multi-core
1	10.6413	11.4553
4	10.2551	4.9785
8	10.5166	3.4352
16	10.4474	2.8926
24	10.1198	2.4787
36	10.5146	2.7698
48	10.6756	2.8742

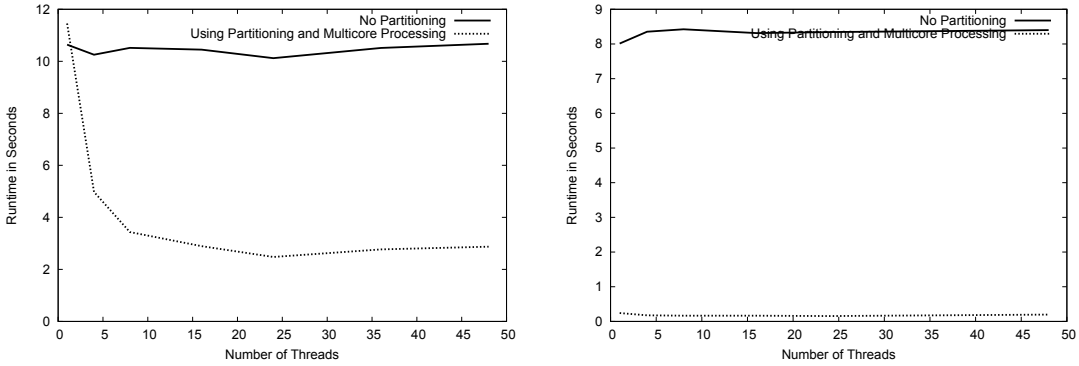
Table 4.5: Indexer Build Run-times using Different Number of Threads.

The results for building the data cube and indexes, presented in Table 4.5, show that partitioning with multi-core processing reaches maximum speedup when the number of threads used is 24, which is twice the number of physical cores available. After this, the processing time increases slightly again. When using 24 threads, the performance increase achieved by the indexer using partitioning and multi-core processing is over 75% compared to the baseline indexer. Figure 4.3(a) shows the build process for both indexers as the number of thread is varied. We also note here that there is a steep increase in performance when the number of threads was increased from 1 to 8, but after that point, the performance only increases slightly.

The results for updating the data cube and indexes are presented in Table 4.6. The

# Threads	Run-times in Seconds	
	No Partitioning	Part. with Multi-core
1	8.0121	0.2421
4	8.3538	0.1757
8	8.4230	0.1653
16	8.3126	0.1663
24	8.3420	0.1550
36	8.3710	0.1765
48	8.3999	0.1965

Table 4.6: Indexer Update Run-times using Different Number of Threads.



(a) Build Process (b) Update Process

Figure 4.3: Indexer Run-times using Varying Number of Threads.

indexer with partitioning and multi-threading shows a minimum performance increase of 97% when compared to the baseline. Optimal performance is reached when the number of threads is equal to the number of physical cores in the system, but the run-time for most thread count is very close. This indicates that the performance increase is achieved by using a "hot" partition and merging updates with this partition and not the entire data cube.

4.2.4 Varying Number of Partitions

In the next set of experiments, we are interested in measuring how partitions affect the indexer. Specifically, we would like to see how the number of partitions affects the indexers with partitioning. Our generated data set consists of 6 dimensions, 1,000,000 tuples, and a skew of 0.1. An update set with 10,000 tuple was also generated.

	Run-times in Seconds	
# Partitions	Partitioning	Part. with Multi-core
1	9.3286	2.2648
2	9.1507	2.1656
4	9.0371	2.1860
8	9.2973	2.4996
16	9.2784	2.2543
32	9.3367	2.5240
64	9.3379	2.5313

Table 4.7: Indexer Build Run-times using Different Number of Initial Partitions.

The results for building the data cube and indexes are shows in Table 4.7. Changing the number of partitions has minimal effect for each indexer. That is, the indexer using only partitioning shows a constant run-time, and the same applies to the indexer using partitioning and multi-core processing. The latter shows a performance increase of least 72% compared to the indexer without multi-core processing.

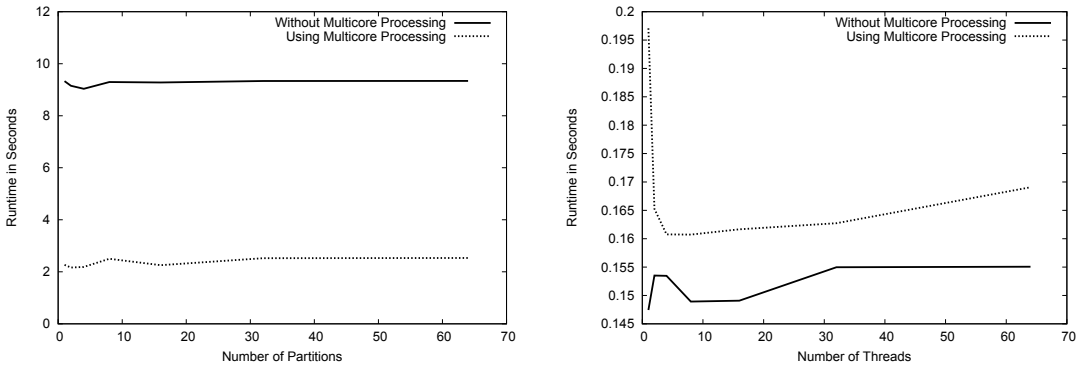
The results for updating the data cube and indexes are presented in Table 4.8. The run-time increases for both indexers, but the indexer using only partitioning outperforms the indexer using both partitioning and multi-core processing.

4.2.5 Varying the Size of the "hot" Partition

The goal of these experiments is to determine the effect the size of the "hot" partition has on indexer run-times when updates are merged with the data cube. We generated

# Partitions	Run-times in Seconds	
	Partitioning	Part. with Multi-core
1	0.1474	0.1971
2	0.1535	0.1653
4	0.1535	0.1608
8	0.1490	0.1607
16	0.1491	0.1617
32	0.1550	0.1627
64	0.1551	0.1691

Table 4.8: Indexer Update Run-times using Different Number of Initial Partitions.



(a) Build Process (b) Update Process

Figure 4.4: Indexer Run-times using Varying Number of Partitions.

	"hot" Partition Size (Run-times in Seconds)			
Update Iteration	6,500	10,000	20,000	50,000
1	0.1648	0.1703	0.1595	0.4152
2	0.3031	0.1631	0.1687	0.6062
3	0.3069	0.1656	0.1686	0.7635
4	0.2972	0.1663	0.1654	0.9178
5	0.3168	0.1689	0.1607	1.0793
6	0.3154	0.1745	0.1675	0.4501
7	0.3219	0.1668	0.1664	0.5816
8	0.3230	0.1693	0.1692	0.7040
9	0.3194	0.1726	0.1663	0.8465
10	0.3194	0.1584	0.1594	0.9711
11	0.3364	0.1585	0.1533	0.3318
12	0.3309	0.1584	0.1664	0.4907

Table 4.9: Non Multi-core Indexer Update Run-times using "hot" Partitions of Various Sizes.

a data set with 6 dimensions, 1,000,000 tuples, and a skew of 0.1. The update set consisted of 20,000 tuples. For multi-core processing, we used 12 threads on a system with 12 physical cores.

The results for updating the data cube and indexes for the indexer using partitioning but no multi-core processing are given in Table 4.9. The best performance was achieved when the update set was multiple of the "hot" partition size. In this case, the "hot" partition sizes of 10,000 and 20,000 provide optimum performance because for every update iteration, the partitions are filled and capped. This means that during the next update iteration, the new "hot" partition is empty, and therefore does not require any merging with the update set. When using a "hot" partition size of 6500, the first iteration of the update set will produce an optimum run-time, but subsequent updates will have decreased performance because the "hot" partition is

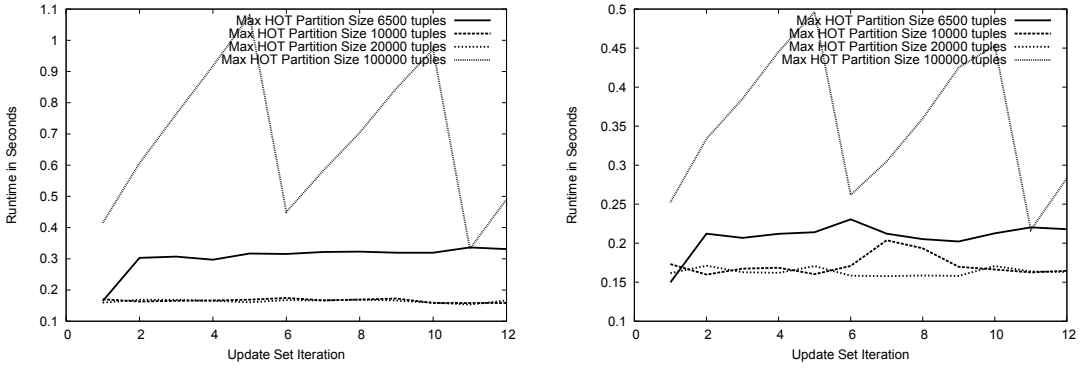
	"hot" Partition Size (Run-times in Seconds)			
Update Iteration	6,500	10,000	20,000	50,000
1	0.1497	0.1732	0.1616	0.2524
2	0.2122	0.1598	0.1711	0.3337
3	0.2068	0.1673	0.1628	0.3850
4	0.2120	0.1686	0.1620	0.4452
5	0.2140	0.1603	0.1706	0.4962
6	0.2305	0.1708	0.1584	0.2619
7	0.2122	0.2038	0.1576	0.3049
8	0.2052	0.1934	0.1586	0.3597
9	0.2022	0.1697	0.1581	0.4252
10	0.2126	0.1663	0.1707	0.4532
11	0.2203	0.1627	0.1638	0.2169
12	0.2179	0.1646	0.1630	0.2833

Table 4.10: Multi-core Indexer Update Run-times using "hot" Partitions of Various Sizes.

not empty.

"hot" partition sizes larger than the update lead to a run-time increase until the partition is capped and a new "hot" partition is started. This is seen in Figure 4.5(a) when then "hot" partition size is set to 100,000 tuples.

The same experiments were carried out using the indexer using partitioning and multi-core processing, and presented in Table 4.10. The results are similar to those obtained for the indexer without multi-core processing, but the overall processing time has increased. We see this when we compare Figure 4.5(a) with Figure 4.5(b). This is consistant with the results of previous experiments, where we see that updating data cubes with 1,000,000 tuples using multi-core processing has a higher run-time than when multi-core is not used.



(a) Without Multi-core Processing (b) Using Multi-core Processing

Figure 4.5: Varying the Size of the "hot" Partition.

4.3 Query Engine

We now turn our attention to query resolution. Here, we are interested in measuring the effect of partitioning on query resolution times, and determining if multi-core processing can help alleviate the extra processing required to merge result sets from multiple partitions, and do duplicate elimination on the fly.

4.3.1 Single Tuple Query

We see the run-time for resolving a query that seeks only one tuple in the data cube. Our test cases use six data cubes and their respective indexes, which were generated using the indexer. The data cubes were generated using data sets with 10,000, 100,000 and 1,000,000 tuples respectively, where three data cubes were non partitioned, and the remaining three were partitioned. The data sets consisted of 6 dimensions, and a 0.1 skew.

Table 4.11 shows the results of running a single tuple query using the query engine against the non partitioned cubes, the partitioned cubes, and against the partioned

	Run-times in Seconds		
Tuples	No Partitioning	Partitioning	Part. with Multi-core
100,000	0.0129	0.0133	0.1262
1,000,000	0.0126	0.0126	0.1252
10,000,000	0.0138	0.0149	0.1096

Table 4.11: Query Engine - 1 Tuple Query.

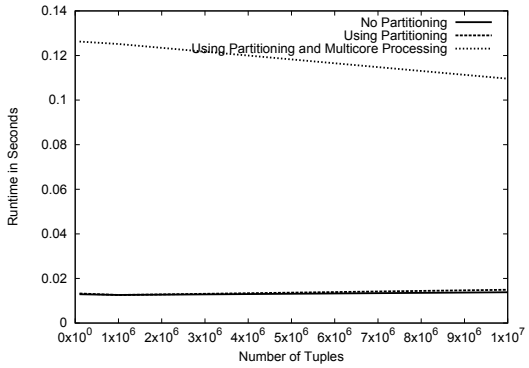


Figure 4.6: Query Returning a Single Tuple.

cube but using multi-core processing. We see that resolving a single tuple leads to very similar run-times for non partitioned and partitioned cubes, but adding multi-core processing leads to an increase in run-time. We see in Figure 4.6 that the run-time for multi-core processing decreases as the size of the data cube increases, but it is still slower than when multi-core processing is not used.

4.3.2 Range Query

Next, we look at the run-time for resolving a range query. We define a range query as a query that returns a set of tuples located that are located close together in the Hilbert space. Our test cases use six data cubes and their respective indexes that were generated using the indexer. The data cubes were generated using data sets with 10,000, 100,000 and 1,000,000 tuples respectively, where three data cubes were

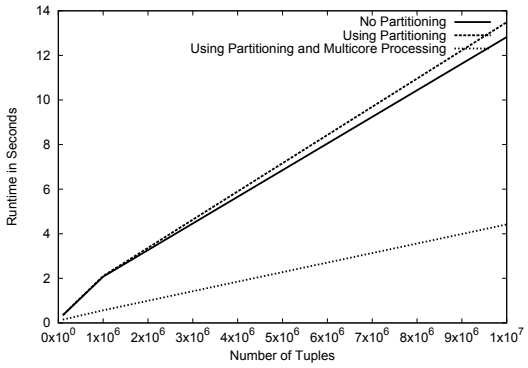


Figure 4.7: Range Query.

non partitioned, and the remaining three were partitioned. The data sets consisted of 6 dimensions, and a 0.1 skew.

	Run-times in Seconds		
Tuples	No Partitioning	Partitioning	Part. with Multi-core
100,000	0.3418	0.3481	0.1486
1,000,000	2.0727	2.1042	0.5685
10,000,000	12.8237	13.4893	4.4163

Table 4.12: Query Engine - Range Query.

The results are presented in Table 4.12. Unlike the single tuple query, we see that the query engine with multi-core processing resolves range queries more quickly than both query engines without multi-core processing. In Figure 4.7, we also see that partitioning leads to slightly higher run-time than non partitioning, when we compare the query engine without partitioning and the query egine with only partitioning. For a data cube with 1,000,000 tuples, the query engine with partitioning and multi-core processing has a performance increase of at least 72%.

	Run-times in Seconds		
Tuples	No Partitioning	Partitioning	Part. with Multi-core
100,000	0.7195	0.8387	0.5685
1,000,000	7.0165	8.3011	5.2573
10,000,000	68.1275	82.3134	55.1430

Table 4.13: Query Engine - Pathologically Large Query.

4.3.3 Pathologically Large Query

In this section, we look at the run-time for resolving a pathologically large query. We define a pathologically large query as a query that returns every tuple or almost every tuple in the data cube. Our test cases use six data cubes and their respective indexes that were generated using the indexer. The data cubes were generated using data sets with 10,000, 100,000 and 1,000,000 tuples respectively, where three data cubes non partitioned, and the remaining three were partitioned. The data sets consisted of 6 dimensions, and a 0.1 skew. Our query returns every tuple in the data cubes.

The results are presented in Table 4.13. In this scenario, the query engine with multi-core processing still performs better than the two other query engines, but the performance increase is not as marked as in the results in Section 4.3.2. For a query resolved against a data cube with 1,000,000 tuples, we only see a 13% decrease in run-time. An explanation for this is the extra time required to merge the partitions and to perform duplicate elimination, as well as large amount of sequential I/O.

4.3.4 Hilbert Packing vs. Hilbert Striping

Finally, we look at the effect of Hilbert packing versus Hilbert striping on the run-time and also on disk performance. In Hilbert packing, we store Hilbert indexes that are spatially close in the same data block on disk. In Hilbert striping, Hilbert indexes that are spatially close are striped across the data blocks used to store the data cube.

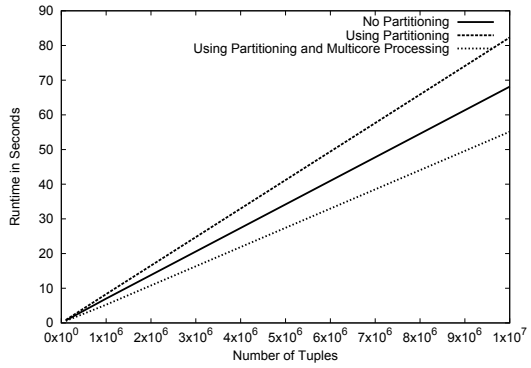


Figure 4.8: Pathologically Large Query.

	Run-times in Seconds	
Tuples	Hilbert Packing	Hilbert Striping
100,000	0.3541	0.4671
1,000,000	2.0887	2.9925
10,000,000	13.3868	17.0336

Table 4.14: Query Engine - Packing vs. Striping - Run-times.

Our test cases use six data cubes and their respective indexes that were generated using the indexer. The data cubes were generated using data sets with 100,000, 1,000,000 and 10,000,000 tuples respectively, where three data cubes were stored using Hilbert packing, and the remaining three using Hilbert striping. The data sets consisted of 6 dimensions, and a 0.1 skew. We used the same range queries from Section 4.3.2, and used the query engine with partitioning but without multi-core processing.

The results with run-times are presented in Table 4.14. We see that when resolving are range query on a data cube with 10,000,000 tuples, Hilbert packing leads to over 21% faster run-time, and Figure 4.9 shows that as the size of the data cube gets larger, Hilbert packing provides better performance.

Next, we measure the number of blocks accessed when resolving the query. The

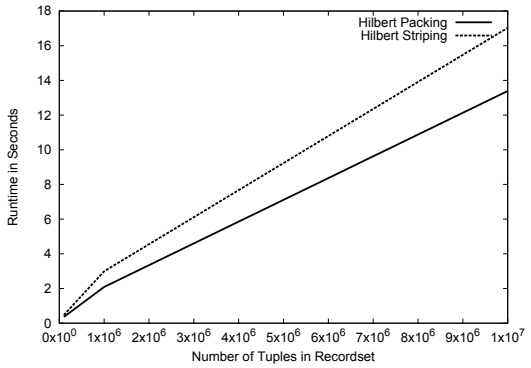


Figure 4.9: Hilbert Packing vs. Hilbert Striping - Run-time.

	Run-times in Seconds	
Tuples	Hilbert Packing	Hilbert Striping
100,000	75	108
1,000,000	419	615
10,000,000	2449	3184

Table 4.15: Query Engine - Packing vs. Striping - Disk Accesses.

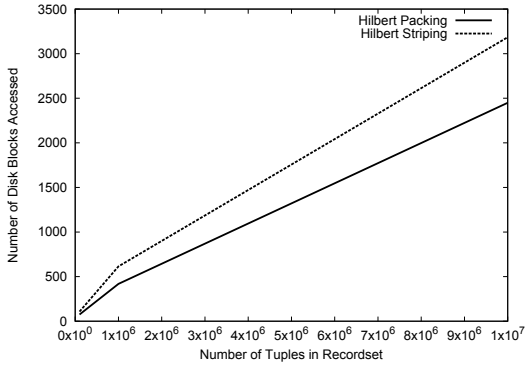


Figure 4.10: Hilbert Packing vs. Hilbert Striping - Disk Accesses.

	Run-times in Seconds	
Tuples	Hilbert Packing	Hilbert Striping
100,000	19	24
1,000,000	82	92
10,000,000	256	624

Table 4.16: Query Engine - Packing vs. Striping - Disk Seeks.

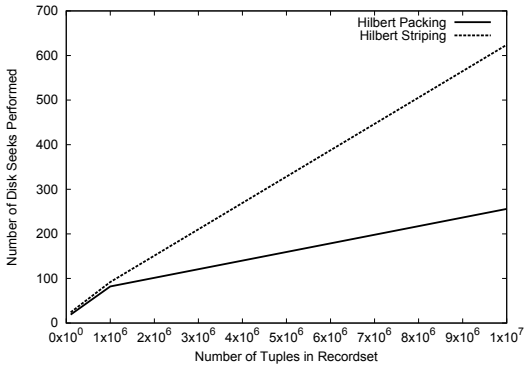


Figure 4.11: Hilbert Packing vs. Hilbert Striping - Disk Seeks.

goal is to access the least number of blocks. The results for disk accesses are presented in Table 4.15. Again, we see that Hilbert packing leads to 23% less data block accessed compared to Hilbert striping, and Figure 4.10 shows that as the size of the data cube gets larger, Hilbert packing provides better performance.

Finally, we measure the number of disk seeks when resolving the query. Disk seeks usually forces the disk head to search for another location on the disk platter, and it is an important factor leading to higher disk latency. Therefore, it is important to minimize disk seeks. The results with disk seeks are presented in Table 4.16. Once again, we see that Hilbert packing leads to 59% less disk seeks when compared to Hilbert striping, and Figure 4.11 shows that as the size of the data cube gets larger, Hilbert packing provides better performance.

Chapter 5

Conclusions

5.1 Summary

To achieve soft real-time analysis in OLAP, the data cube must be updated frequently to reflect the most current state of the fact table in the DW, without negatively affecting query resolution time or taking the system offline. In this thesis, we have provided a framework that implements local partitioning and maintains a "hot" partition to accelerate data cube updating. It also leverages multi-core processing to improve runtimes for building the data cube, and query resolution, thereby improving the system performance for users while updates are being applied. The following is a summary of the key improvements our framework has added to the Sidera server that has enabled soft real-time OLAP.

- **Soft Real-time Updates**

  Compared to baseline measurements using an indexer that does not generate partitioned views, our framework shows over 99% in performance increase when updating a data cube of 10,000,000 tuples with an update set of 100,000 tuples. We also noted that as the size of the data cubes and the update sets became

larger, the performance gain obtained using our framework increased. This allows updates from the DW to be merged more frequently, thereby providing users with data cubes that contain more up-to-date information.

Our experiments also indicated that multi-core processing had less effect on the update process, but rather that the "hot" partitioning strategy was responsible for the performance improvements because it limits the number of tuples from the existing data cube that must be decompressed for merging. When the data cube and the update sets are small, multi-core processing is slower than the indexer without multi-core processing. As the number of tuples grows, the benefits of parallel processing begin to outweigh the overhead of managing multiple threads.

- **Faster Data Cube Construction**

  Data cube construction benefits from multi-core processing because of the high volume of Hilbert conversions. When building a data cube containing 10,000,000 tuples, our framework performed 76% faster than the baseline measurements. Like the update process, the performance difference increases as the size of the data cube becomes larger. Partitioning the data cube adds an overhead to performance because mutiple compressed data and index files must be created.

  We have also noted that when using multi-core processing, we see a steep gain in performance when increasing the number of cores allocated to the process from 1 to 8 cores, but after this point, the gains were less significant.

- **Faster Query Resolution**

  We tested our framework using three query scenarios, namely the single tuple

query, the range query and the pathologically large query. Of the three, the most common type of query issued by users is the range query. The single tuple query shows no marked difference in performance between the baseline and the indexer implementing partitioning but without multi-core processing. Multi-core processing had a negative impact on single tuple queries because the overhead of managing multiple threads outweighs the performance gains.

Our framework provides a performance increase of at least 72% over the baseline measurements when resolving a range query against a data cube with 1,000,000 tuples. Finally, the framework also performed better in pathologically large queries, but since sequential I/O becomes the dominant cost, the performance increase was limited to 25%.

- **Multi-core Processing**

  Multi-core processing was added incrementally using the OpenMP library to both the indexer and query engine. The greatest benefits from the multi-core processing is seen when the initial data cube is generated by the indexer, as well when resolving range queries.

- **More Efficient I/O**

  I/O is a bottleneck in application performance. In our framework, the I/O is optimized to perform the least number of disk seeks by reading and write sequential blocks. This was done by developing an in-memory buffering subsystem, replacing the previous file based buffering subsystem. While the distributed-memory Sidera server used Hilbert striping in order to achieve optimum load

balance across the cluster, our framework uses Hilbert packing, which minimizes the number disk blocks retrieved during range queries, thereby reducing the runtime.

5.2 Future Work

As a final thought, we propose a few topics for future work based on the framework presented in this thesis. These were not included in our research due to either time constraints or lack of necessary hardware required for testing. We nevertheless feel that these can be interesting avenues of research to improve the performance of OLAP tools, specifically a server such as Sidera.

- **Improving I/O Performance**

  Althouth it is possible to do parallel I/O by using barriers to ensure that only one thread reads or writes to a file, it is nevertheless a self-defeating purpose. In our architecture, we have separated the I/O output away from the other stages, and read and write operations are done as sequential operations. This reduce significantly the I/O time, which is probably the largest bottleneck in a system such as Sidera.

  Hardware assisted parallel I/O can assist in performing read and write operations in parallel while minimizing disk seeks. This would allow multiple views to be processed in parallel during the build and update process. When resolving queries, blocks from multiple files could be read simultaneously.

- **Integrating Shared-memory OLAP Architecture with distributed-memory architecture**

Our framework is constructed to enable parallel processing in Sidera on a single machine, taking advantage of modern multi-core processors. To take full advantage of parallel processing, our framework should be integrated into the distributed Sidera server described in Section 3.2. Views would be partitioned at the cluster level and using the PSI, they would striped and distributed across the nodes. Each node would then locally partition the tuples further, and generate HTDC compressed views and indexes.

With the approach, the architecture would offer three levels of parallelism. Each node would offer multiple cpus and each cpu would have multiple cores. A finally, the cluster would offer multiple nodes.

- **Improving Work Distribution across Cores**

  As the experimental results in Section 4.2.3 indicate, near maximum performance for the build process is reached when aproximately 8 threads running on 8 cores perform the build process. After this point, dedicating more theads/cores do not significantly improve the performance and could be utilized better when performing other tasks. For systems with more than 8 physical cores, an optimization to our framework would be to process two or more separate materialized view simultaneously. Another optimization can include dedicating a specific number of cores for data cube updates, and other cores for query resolution.

Bibliography

[1] R. Agrawal, A. Ailamaki, P. A. Bernstein, E. A. Brewer, M. J. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. J. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. F. Korth, D. Kossmann, S. Madden, R. Magoulas, B. C. Ooi, T. O'Reilly, R. Ramakrishnan, S. Sarawagi, M. Stonebraker, A. S. Szalay, and G. Weikum. The Claremont Report on Database Research. *Commun. ACM*, 52(6):56–65, 2009.

[2] S. Agrawal, V. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 359–370, New York, NY, USA, 2004. ACM.

[3] M. Ahuja, C. Chen, R. Gottapu, J. Hallmann, W. Hasan, R. Johnson, M. Kozyrczak, R. Pabbati, N. Pandit, S. Pokuri, and K. Uppala. Peta-scale Data Warehousing at Yahoo! In *SIGMOD'09: Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 855–862, New York,NY,USA, 2009. ACM.

[4] G. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint*

Computer Conference, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[5] M. H. Bateni, L. Golab, M. T. Hajiaghayi, and H. Karloff. Scheduling to Minimize Staleness and Stretch in Real-time Data Warehouses. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 29–38, New York, NY, USA, 2009. ACM.

[6] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[7] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD'90: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, New York,NY,USA, 1990. ACM.

[8] J. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18:509–517, September 1975.

[9] D. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[10] Ying C., A. Rau-Chaplin, F. Dehne, T. Eavis, D. Green, and E. Sithirasenan. cgmOLAP: Efficient Parallel Generation and Querying of Terabyte Size ROLAP Data Cubes. In *ICDE*, page 164, 2006.

[11] B. Chapman, A. Patil, and A. Prabhakar. Performance Oriented Programming for NUMA Architectures. In *Proceedings of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, WOMPAT '01, pages 137–154, London, UK, 2001. Springer-Verlag.

[12] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[13] Y. Chen, F. Dehne, T. Eavis, and A. Rau-Chaplin. Parallel ROLAP Data Cube Construction on Shared-Nothing Multiprocessors. In *Proc. International Parallel and Distributed Processing Symposium*, 2003.

[14] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13:377–387, June 1970.

[15] IBM Corporation. X10: Performance and Productivity at Scale. http://x10-lang.org/, Last Accessed March 12, 2012.

[16] IBM Corporation. IBM DB2 Database for Linux, UNIX, and Windows Information Center. http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp, Last Accessed March 15, 2012.

[17] Intel Corporation. Intel Processor Comparison. http://www.intel.com, Last Accessed February 14, 2012.

[18] Microsoft Corporation. SQL Server 2008 R2 - Partitioned Tables and Indexes. http://msdn.microsoft.com/en-us/library/ms188706.aspx, Last Accessed March 15, 2012.

[19] Oracle Corporation. Oracle Partitioning. http://www.oracle.com/us/products/database/options/partitioning/index.html, Last Accessed March 15, 2012.

[20] U. Dayal, M. Castellanos, A. Simitsis, and K. Wilkinson. Data Integration Flows for Business Intelligence. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 1–11, New York, NY, USA, 2009. ACM.

[21] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the Data Cube. *Distrib. Parallel Databases*, 11:181–201, March 2002.

[22] F. Dehne, T. Eavis, and A. Rau-Chaplin. Coarse Grained Parallel On-Line Analytical Processing (OLAP) for Data Mining. In *Proceedings of the International Conference on Computational Science-Part II*, ICCS '01, pages 589–598, London, UK, UK, 2001. Springer-Verlag.

[23] F. Dehne, T. Eavis, and A. Rau-Chaplin. Computing Partial Data Cubes for Parallel Data Warehousing Applications. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 319–326, London, UK, 2001. Springer-Verlag.

[24] F. Dehne, T. Eavis, and A. Rau-Chaplin. Top-down Computation of Partial ROLAP Data Cubes. In *Proc. 37th Annual Hawaii International Conference on System Sciences*, 2004.

[25] F. Dehne, T. Eavis, and A. Rau-Chaplin. Efficient Computation of View Subsets. In *DOLAP'07: Proceedings of the ACM Tenth International Workshop on Data Warehousing and OLAP*, pages 65–72, New York,NY,USA, 2007. ACM.

[26] D. Düllmann. Petabyte Databases. *SIGMOD Rec.*, 28:506, June 1999.

[27] T. Eavis and D. Cueva. A Hilbert Space Compression Architecture for Data Warehouse Environments. In *DaWaK*, pages 1–12, 2007.

[28] T. Eavis and D. Cueva. The LBF R-tree: Efficient Multidimensional Indexing with Graceful Degradation. In *Proc. 11th International Database Engineering and Applications Symposium IDEAS 2007*, pages 241–250, 2007.

[29] T. Eavis, G. Dimitrov, I. Dimitrov, D. Cueva, A. Lopez, and A. Taleb. Sidera: A Cluster-based Server for Online Analytical Processing. In *Proceedings of the 2007 OTM Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part II*, OTM'07, pages 1453–1472, Berlin, Heidelberg, 2007. Springer-Verlag.

[30] T. Eavis and A. Lopez. Rk-hist: An R-tree Based Histogram for Multi-Dimensional Selectivity Estimation. In *CIKM'07: Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, pages 475–484, New York,NY,USA, 2007. ACM.

[31] T. Eavis and R. Sayeed. High Performance Analytics with the R3-Cache. In *Proceedings of the 11th International Conference on Data Warehousing and Knowledge Discovery*, DaWaK '09, pages 271–286, Berlin, Heidelberg, 2009. Springer-Verlag.

[32] T. Eavis and A. Taleb. Mapgraph: Efficient Methods for Complex OLAP Hierarchies. In *CIKM'07: Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, pages 465–474, New York,NY,USA, 2007. ACM.

[33] R. Finkel and J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.*, 4:1–9, 1974.

[34] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[35] M.J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[36] L. Golab, T. Johnson, and V. Shkapenyuk. Scheduling Updates in a Real-Time Stream Warehouse. In *IEEE 25th International Conference on Data Engineering, 2009. ICDE '09.*, pages 1207–1210, April 2009.

[37] L. Golab, T. Johnson, and V. Shkapenyuk. Scalable Scheduling of Updates in Streaming Data Warehouses. *IEEE Transactions on Knowledge and Data Engineering*, PP(99):1, 2011.

[38] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By,Cross-Tab,and Sub-Totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[39] Khronos Group. OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems. http://www.khronos.org/opencl/, Last Accessed March 15, 2012.

[40] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, New York,NY,USA, 1984. ACM.

[41] F.G. Heath. Origins of the Binary Code. *Scientific American*, pages 76–83, August 1972.

[42] D. Hilbert. Über die stetige Abbildung einer Line auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891. 10.1007/BF01199431.

[43] W. Inmon. *Building the Data Warehouse*. John Wiley & Sons, Inc., New York, NY, USA, 2005.

[44] H. V. Jagadish. Linear Clustering of Objects with Multiple Attributes. *SIGMOD Rec.*, 19:332–342, May 1990.

[45] I. Kamel and C. Faloutsos. On Packing R-trees. In *CIKM'93: Proceedings of the Second International Conference on Information and Knowledge Management*, pages 490–499, New York,NY,USA, 1993. ACM.

[46] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In *VLDB'94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 500–509, San Francisco,CA,USA, 1994. Morgan Kaufmann Publishers Inc.

[47] H. Karloff, S. Suri, and S. Vassilvitskii. A Model of Computation for MapReduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 938–948, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[48] R. Kimball and J. Caserta. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. Wiley, 2004.

[49] R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. Safari Books Online. Wiley, 2002.

[50] Labio, W. J. and Yerneni, R. and Garcia-Molina, H. Shrinking the Warehouse Update Window. *SIGMOD Rec.*, 28:383–394, June 1999.

[51] C. Li, W. Rahayu, and D. Taniar. Towards Near Real-Time Data Warehousing. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference*, pages 1150–1157, 2010.

[52] Quinn M. *Parallel Programming in C With MPI and OpenMP*. Mcgraw Hill Higher Education, September 2003.

[53] Argonne National Laboratory Mathematics and Computer Science Division. The Message Passing Interface (MPI) Standard. http://www.mcs.anl.gov/research/projects/mpi/, Last Accessed March 15, 2012.

[54] B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, jan/feb 2001.

[55] D. Moore. Fast Hilbert Curve Generation, Sorting, and Range Queries. `http://www.tiac.net/~sw/2008/10/Hilbert/moore/hilbert.c`, Last Accessed February 14, 2012.

[56] G. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8), April 1965.

[57] G.M. Morton. A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. 1966.

[58] W. K. Ng and C. V. Ravishankar. Block-Oriented Compression Techniques for Large Statistical Databases. *IEEE Trans. on Knowl. and Data Eng.*, 9:314–328, March 1997.

[59] G. Peano. Sur Une Courbe Qui Remplit Toute Une Aire Plane. *Mathematishe Annalen*, 36:57–160, 1890.

[60] John T. Robinson. The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, SIGMOD '81, pages 10–18, New York, NY, USA, 1981. ACM.

[61] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and Bulk Incremental Updates on the Data Cube. In *SIGMOD'97: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 89–99, New York,NY,USA, 1997. ACM.

[62] N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases using Packed R-trees. In *SIGMOD'85: Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, pages 17–31, New York,NY,USA, 1985. ACM.

[63] R. J. Santos and J. Bernardino. Real-time Data Warehouse Loading Methodology. In *Proceedings of the 2008 international symposium on Database Engineering and Applications*, IDEAS '08, pages 49–58, New York, NY, USA, 2008. ACM.

[64] R. J. Santos, J. Bernardino, and M. Vieira. 24/7 Real-Time Data Warehousing: A Tool for Continuous Actionable Knowledge. In *Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference*, COMPSAC '11, pages 279–288, Washington, DC, USA, 2011. IEEE Computer Society.

[65] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB'87: Proceedings of the 13th International Conference on Very Large Data Bases*, pages 507–518, San Francisco,CA,USA, 1987. Morgan Kaufmann Publishers Inc.

[66] Y. Sismanis, A. Deligiannakis, Y. Kotidis, and N. Roussopoulos. Hierarchical Dwarfs for the Rollup Cube. In *DOLAP'03: Proceedings of the 6th ACM International Workshop on Data Warehousing and OLAP*, pages 17–24, New York,NY,USA, 2003. ACM.

[67] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the PetaCube. In *SIGMOD'02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 464–475, New York,NY,USA, 2002. ACM.

[68] A. S. Szalay. Scientific Publishing in the Era of Petabyte Data. In *Proceedings of the 8th ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '08, pages 261–262, New York, NY, USA, 2008. ACM.

[69] The OpenMP Architecture Review Board. The OpenMP API Specification for Parallel Programming. http://openmp.org, Last accessed March 15, 2012.

[70] C. Thomsen and T. B. Pedersen. Easy and Effective Parallel Programmable ETL. In *Proceedings of the ACM 14th International Workshop on Data Warehousing and OLAP*, DOLAP '11, pages 37–44, New York, NY, USA, 2011. ACM.

[71] UC Berkeley/LBNL. Berkeley UPC - Unified Parallel C. http://upc.lbl.gov/, Last Accessed March 15, 2012.

[72] C. Ying, F. Dehne, T. Eavis, and A. Rau-Chaplin. Building Large ROLAP Data Cubes in Parallel. In *IDEAS*, pages 367–377, 2004.