PEDRO EUGÊNIO ROCHA PEDREIRA

ON INDEXING HIGHLY DYNAMIC MULTIDIMENSIONAL DATASETS FOR INTERACTIVE ANALYTICS

CURITIBA

2016

PEDRO EUGÊNIO ROCHA PEDREIRA

ON INDEXING HIGHLY DYNAMIC MULTIDIMENSIONAL DATASETS FOR INTERACTIVE ANALYTICS

Submitted to the Department of Informatics in partial fulfillment of the requirements for the degree of Ph.D in Informatics at the Federal University of Paraná - UFPR.
Advisor: Prof. Dr. Luis Carlos Erpen de Bona

CURITIBA

2016

Pedreira, Pedro Eugênio Rocha
On indexing highly dynamics multidimensional datasets for interactive analytics / Pedro Eugênio Rocha Pedreira. – Curitiba, 2016
91 f. : il.; tabs.

Thesis (Degree of Ph.D) – Federal University Of Paraná, Exact Science Sector, Graduate Program in Informatics.
. Advisor: Luis Carlos Erpen de Bona
Bibliografy: p. 77-91

1. Indexing. 2. Data Storage. 3. Database – Management. I. Bona, Luis Carlos Erpen de. II. Título

CDD 005.74

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
Setor CIÊNCIAS EXATAS
Programa de Pós Graduação em INFORMÁTICA
Código CAPES: 40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Tese de Doutorado de **PEDRO EUGENIO ROCHA PEDREIRA**, intitulada: "**ON INDEXING HIGHLY DYNAMIC MULTIDIMENSIONAL DATASETS FOR INTERACTIVE ANALYTICS**", após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua APROVAÇÃO.

Curitiba, 15 de Abril de 2016.

Prof LUIS CARLOS ERPEN DE BONA
Presidente da Banca Examinadora (UFPR)

Prof EDUARDO CUNHA DE ALMEIDA
Avaliador Interno (UFPR)

Prof MARCO ANTONIO ZANATA ALVES
Avaliador Externo (UFPR)

Prof FABIO ANDRE MACHADO PORTO
Avaliador Externo (LNCC)

Prof GUSTAVO ALBERTO GIMENEZ LUGO
Avaliador Externo (UTFPR)

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

To my mom Maria and wife Gaby, and all of those who

helped me keeping my sanity in the last five years.

You all did a partially good job.

ACKNOWLEDGMENTS

I would like to thank all members of the Cubrick Team at Facebook and all the great Engineers that in any ways contributed to the project since its inception. A big shout out to Amit Dutta, Yinghai Lu, Sergey Pershin, Guangdeng Liao, Elon Azoulay, Justin Palumbo, Senthil Babu Jegannathan and many others. I also want to acknowledge Jason Sundram, Burc Arpat and Guy Bayes for believing and supporting the pretentious idea of building a new DBMS from scratch. Special thanks to Chris Croswhite who was much more than a manager in the last few years, and vastly helped me in multiple technical and personal aspects — this work would not be possible without your help. Lastly, thanks to Facebook for being such an open environment and for founding the project.

I would also like to express my gratitude to all the professors and colleagues at UFPR who taught me everything I needed to know about computer science, in special to Prof. Marcos Castilho for the opportunity to join C3SL and Prof. Eduardo Cunha for the insightful database conversations. Above all, my most sincere gratitude to Prof. Luis Bona for the continuous support and guidance over the last decade; I could not have imagined having a better advisor and mentor for my Ph.D, Master's and undergrad studies.

A special thanks to all my family. Words cannot express how grateful I am to my mother for always supporting me, to my father for all the good conversations, to my sister and grandfather (in memory). At the end I would like express my gratitude to my beloved wife Gaby for standing beside me throughout my career and studies. You have been my inspiration and motivation and I am certain I would have never got in here if it wasn't for you. Love you all.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

BAT	*Binary Association Table*
CPU	*Central Processing Unit*
CSV	*Comma-Separated Values*
BESS	*Bit-Encoded Sparse Structure*
BSP-Tree	*Binary Space Partitioning Tree*
DBA	*Database Administrator*
DBMS	*Database Management System*
DBS	*Database System*
GIS	*Geographic Information Systems*
HDFS	*Hadoop Distributed Filesystem*
I/O	*Input/Output*
JSON	*Java Script Object Notation*
LSM	*Log-Structured Merge-Tree*
MBR	*Minimum Bounding Rectangle*
MVCC	*Multiversion Concurrency Control*
OLAP	*Online Analytical Processing*
OLTP	*Online Transaction Processing*
OS	*Operating System*
RAM	*Random Access Memory*
RLE	*Run Length Encoding*
ROLAP	*Relational Online Analytical Processing*
ROS	*Read Optimized Store*
RPC	*Remote Procedure Call*
SGBD	*Sistema de Gerenciamento de Banco de Dados*
SIMD	*Single Instruction Multiple Data*
SLA	*Service Level Agreement*
SQL	*Structured Query Language*
STL	*Standard Template Library*
WOS	*Write Optimized Store*

RESUMO

Indexação de dados multidimensionais tem sido extensivamente pesquisada nas últimas décadas. Neste trabalho, um novo workload OLAP identificado no Facebook é apresentado, caracterizado por (a) alta dinamicidade e dimensionalidade, (b) escala e (c) interatividade e simplicidade de consultas, inadequado para os SGBDs OLAP e técnicas de indexação de dados multidimensionais atuais. Baseado nesse caso de uso, uma nova estratégia de indexação e organização de dados multidimensionais para SGBDs em memória chamada *Granular Partitioning* é proposta. Essa técnica extende a visão tradicional de partitionamento em banco de dados, particionando por invervalo todas as dimensões do conjunto de dados e formando pequenos blocos que armazenam dados de forma não-ordenada e esparsa. Desta forma, é possível atingir altas taxas de ingestão de dados sem manter estrutura auxiliar alguma de indexação. Este trabalho também descreve como um SGBD OLAP capaz de suportar um modelo de dados composto por cubos, dimensões e métricas, além de operações como *roll-ups*, *drill-downs* e *slice and dice* (filtros) eficientes pode ser construído com base nessa nova técnica de organização de dados. Com objetivo de validar experimentalmente a técnica apresentada, este trabalho apresenta o Cubrick, um novo SGBD OLAP em memória distribuído e otimizado para a execução de consultas analíticas baseado em Granular Partitioning, escrito desde a primeira linha de código para este trabalho. Finalmente, os resultados de uma avaliação experimental extensiva contendo conjuntos de dados e consultas coletadas de projetos pilotos que utilizam Cubrick é apresentada; em seguida, é mostrado que a escala desejada pode ser alcançada caso os dados sejam organizados de acordo com o Granular Partitioning e o projeto seja focado em simplicidade, ingerindo milhões de registros por segundo continuamente de fluxos de dados em tempo real, e concorrentemente executando consultas com latência inferior a 1 segundo.

ABSTRACT

Indexing multidimensional data has been an active focus of research in the last few decades. In this work, we present a new type of OLAP workload found at Facebook and characterized by (a) high dynamicity and dimensionality, (b) scale and (c) interactivity and simplicity of queries, that is unsuited for most current OLAP DBMSs and multidimensional indexing techniques. To address this use case, we propose a novel multidimensional data organization and indexing strategy for in-memory DBMSs called *Granular Partitioning*. This technique extends the traditional view of database partitioning by range partitioning every dimension of the dataset and organizing the data within small containers in an unordered and sparse fashion, in such a way to provide high ingestion rates and indexed access through every dimension without maintaining any auxiliary data structures. We also describe how an OLAP DBMS able to support a multidimensional data model composed of cubes, dimensions and metrics and operations such as roll-up, drill-down as well as efficient slice and dice (filtering) can be built on top of this new data organization technique. In order to experimentally validate the described technique we present Cubrick, a new in-memory distributed OLAP DBMS for interactive analytics based on Granular Partitioning we have written from the ground up at Facebook. Finally, we present results from a thorough experimental evaluation that leveraged datasets and queries collected from a few pilot Cubrick deployments. We show that by properly organizing the dataset according to Granular Partitioning and focusing the design on simplicity, we are able to achieve the target scale and store tens of terabytes of in-memory data, continuously ingest millions of records per second from realtime data streams and still execute sub-second queries.

CHAPTER 1

INTRODUCTION

Realtime analytics over large datasets has become a widespread need across most internet companies. Minimizing the time gap between data production and data analysis enables data driven companies to generate insights and make decisions in a more timely manner, ultimately allowing them to move faster. In order to provide realtime analytics abilities, a DBS needs to be able to continuously ingest streams of data — commonly generated by web logs — and answer queries only a few seconds after the data has been generated. The ingestion of these highly dynamic datasets becomes even more challenging at scale, given that some realtime streams can emit several million records per second.

In order to quickly identify trends and generate valuable insights, data analysts also need to be able to *interactively* manipulate these realtime datasets. Data exploration can become a demanding asynchronous activity if each interaction with the DBS layer takes minutes to complete. Ideally, all database queries should finish in hundreds of milliseconds in order to provide a truly interactive experience to users [1]; unfortunately, scanning large datasets in such a short time frame requires massive parallelization and thus a vast amount of resources. For instance, 20,000 CPUs are required in order to scan 10 TB of uncompressed in-memory data to answer a single query in 100 milliseconds[1]. Likewise, reading all data from disk at query time becomes infeasible considering the tight latency and scale requirements.

However, over the last few years of providing database solutions at Facebook we have empirically observed that even though the raw dataset is large, most queries are heavily filtered and interested in a very particular subset of it. For example, a query might only be interested in one metric for a particular demographic, such as only people living in the US or from a particular gender, measure the volume of conversation around a particular

[1]This is a rough extrapolation based on the amount of in-memory data a single CPU is able to scan, on average, in our environment (see Chapter 6).

topic, in a specific group or mentioning a particular hashtag. Considering that the filters applied depend on which aspects of the dataset the analyst is interested, they are mostly ad-hoc, making traditional one-dimensional pre-defined indexes or sort orders less effective for these use cases.

Nevertheless, the requirements of low latency indexed queries and highly dynamic datasets conflict with the traditional view that OLAP DBSs are batch oriented systems that operate over mostly static datasets [2]. It is also well known that traditional row-store DBMSs are not suited for analytics workloads since all columns need to be materialized when looking up a particular record [3]. We argue that even column oriented DBMSs do not perform well in highly dynamic workloads due to the overhead of maintaining a sorted view of the dataset (for C-STORE based architectures [4]) or of keeping indexing structures up-to-date in order to efficiently answer ad-hoc queries. Moreover, current OLAP DBSs are either based on a relational DBMSs (ROLAP) [5] [6] and therefore suffer from the same problems, or are heavily based on pre-aggregations [7] and thus are hard to update. We advocate that a new in-memory DBMS engine is needed in order to conform to these requirements.

In this work we describe a novel indexing technique for in-memory DBMSs that enables the organization of highly dynamic multidimensional datasets, which we call *Granular Partitioning*. This technique extends the traditional notion of database partitioning and allows filters to be efficiently applied over any dimension of the dataset. Assuming that the cardinality of each dimension is known beforehand, granular partitioning range partitions every dimension of the dataset and composes a set of small containers. Hence, by properly numbering the containers their *ids* can be used to prune data at query time and to efficiently locate the target container for a particular record. This organization also allows the engine to store the dimension values relative to the container (instead of the absolute value), resulting in lower memory footprint. Finally, unlike other multidimensional indexing strategies, granular partitioning provides constant time complexity for insertions of new records and constant memory usage.

In addition, leveraging the granular partitioning indexing technique proposed we built

Cubrick, a full-featured distributed in-memory multidimensional DBMS. Cubrick allows the execution indexed OLAP operations such as slice-n-dice, roll-ups and drill-down over very dynamic multidimensional datasets composed of cubes, dimensions and metrics. Typical use cases for Cubrick include loads of large batches or continuous streams of data for further execution of low latency analytic operations, that in general produces small result sets to be consumed by interactive data visualization tools. In addition to low level data manipulation functions, the system also exposes a more generic SQL interface.

Data in a Cubrick cube is range partitioned by every dimension, composing a set of data containers called *bricks* where data is stored sparsely, column-wise and in an unordered and append-only fashion. Each brick is also row-major numbered, considering that the cardinality of each dimension and range size are estimated beforehand, providing fast access to the target brick given an input record and the ability to easily prune bricks out of a query scope given a brick id and a set of ad-hoc filters. Each brick in a Cubrick cube is assigned to a cluster node using consistent hashing [8] and data is hierarchically aggregated in parallel at query time.

1.1 Contributions

In this work we make the following contributions:

- We describe a new class of workloads found by analyzing a few Facebook datasets that has unique characteristics and is unsuited for most current DBS architectures.
- We describe a novel technique able to index and organize highly dynamic multidimensional datasets called Granular Partitioning, that extends traditional database partitioning and supports indexed access through every dimension.
- We present a new DBMS written from the ground up that leverages granular partitioning, detailing the basic data structures that store data in memory and describe details about its query execution engine and distributed architecture.
- We outline how Cubrick's data model and internal design allows us to incremen-

tally accumulate data from different data sources and granularities in a transparent manner for queries.

- We present a thorough experimental evaluation of Cubrick, including a cluster running at the scale of several terabytes of in-memory data, ingesting millions of records per second and executing sub-second queries.

Cubrick is currently being used in production deployments inside Facebook for more than a year, storing several terabytes of in-memory data and spanning hundreds of servers.

1.2 Organization

The remainder of this work is organized as follows. Chapter 2 presents an overview about DBS architectures and the indexing methodologies commonly used. Chapter 3 focuses on both one-dimensional and multidimensional indexing data structures and their use within DBS. In Chapter 4, we start by describing the target workload both in terms of data and queries and how it differentiates from traditional OLAP workloads. We then describe the novel Granular Partitioning technique and point to related work. In Chapter 5 we present Cubrick, its distributed model and details about query processing and rollups. In this chapter we also present an evaluation of the impact of data distribution in the system. Chapter 6 presents a comprehensive experimental analysis of our current implementation, comprising several aspects of the Cubrick server. Finally, Section 7 concludes this thesis.

CHAPTER 2

DATABASE ARCHITECTURES

The relational model, as first proposed by Codd [9] in the early 70s, is inherently two-dimensional. In this model, data is organized in terms of one-dimensional *tuples* containing one or more attributes, and further grouped into unordered sets, called *relations*. One can think of the first dimension of a relation as being its different tuples and the second being attributes inside a particular tuple.

In order to map these two-dimensional conceptual abstractions into physical storage devices that only expose a one-dimensional interface, such as disks or even RAM, traditional relational database management systems have two obvious choices: store data row-by-row or column-by-column. Historically, these systems have focused their efforts on a row-oriented storage model since they usually perform better in business transactional workloads; nevertheless, in the last decade, column-oriented databases management systems have become more popular [10] as a new class of workloads composed of analytical operations have become increasingly important.

Column-oriented database management systems (also called *column-stores*) have very unique characteristics that differentiate them from row-oriented database management systems (or *row-stores*), thus requiring a complete redesign not only on the storage layer, but also in the query execution and query optimizer subsytems [3]. In fact, in addition to the academic claim that a new design was required for column-stores [11], the most commercially successful column-stores nowadays are complete rewrites [12] [13], rather than classic row-stores with a column-oriented storage layer.

Throughout this thesis we follow the terminology in [14], which defines that a *database* is a collection of related data, a *Database Management System* (DBMS) is the collection of software that enables a user to create and manipulate a database, and a *database system* (DBS) is the broader term that encompasses the entire database and DBMS ecosystem.

This remainder of this chapter is organized as follows. We start by describing the overall data layout of row-store DBMSs in Section 2.1 and column-store DBMSs in Section 2.2. In Section 2.3 we present the different methodologies that can be followed when deciding what to index in a database, while Section 2.4 describes the main concepts about database partitioning.

2.1 Row-Oriented Databases

Row-oriented DBMSs store data row-by-row, or one record at a time. Since attributes of a record are stored contiguously, this organization is well suited for workloads that usually access data in the granularity of an entity. For instance, in a database that keeps information about users of a particular system, a row-store DBMS can efficiently execute operations such as find a user, add a new user and delete or update an existing user. DBSs designed to support this type of workload, also known as OLTP (*Online Transaction Processing*), are usually optimized towards executing small operations with high transactional throughput, rather than efficiently supporting a single large transaction.

One drawback usually found in row-store DBSs relies in the fact that disk storage interfaces are block oriented. That means that the smallest granularity in which data can be read from disk is a block, which depending on the schema definition can comprise a handful of records. Therefore, even if a query only requires access to one column of a record, usually the entire record needs to be read from disk, or at least all data sharing the same disk block. For queries that need to access one column of each single record in a database (to calculate the average age of all users, for instance), a lot of disk I/O can be wasted reading unused data. In addition, since disk I/O throughput is usually higher when transferring larger sequential blocks, DBMSs tend to increase the block size in order to make efficient use of I/O bandwidth. The same assertions still hold for in-memory row-store DBMSs, since reading entire records when only one column is needed causes more cache misses and increases the traffic between memory and CPU.

2.2 Column-Oriented Databases

Unlike row-store DBMSs, column-oriented systems completely partition the database vertically, storing data one column at a time. This organization allows the DBMS to read only the columns needed by a particular query instead of reading entire records and discarding unneeded data, hence using disk I/O more efficiently. Column-stores are well suited for analytical workloads where queries need to access and aggregate values for a few columns at a time, but scanning billions of records. Besides, these DBMSs are usually composed by wide denormalized tables following a multidimensional schema (either *star* or *snow-flake* [2]), where tables can contain up to hundreds of columns; in these cases, reading the entire table (all columns) per query is not a viable solution. However, despite being able to execute analytical queries efficiently, column-store DBMSs perform poorly under transactional workloads that require access to several columns at a time of only a few records, since usually one disk operation needs to be issued to read each different column.

As a result of how column-oriented DBMSs store data on disk, several interesting techniques can be leveraged in order to optimize resource utilization and lower query latency:

- **Column-specific encoding.** By definition, all data stored in a particular column has the same data type, what allows the implementation of efficient encoding techniques. Data encoding reduces the disk space needed to store column files, and consequently lowers the amount of disk I/O needed to transfer them. Therefore, encoding increases the overall DBS performance, considering that disk I/O throughput is usually a bottleneck [11]. A few examples of column encoding are:
  - *Delta Encoding.* Stores deltas between consecutive values instead of the values themselves. Good for columns like primary keys and other ordered sets of integer values.
  - *Run-Length Encoding (RLE).* For ordered low cardinality columns, a sequence of repeated values can be represented by a tuple containing the value and the

number of repetitions [15].

  - *Dictionary Encoding.* Dictionary encoding keeps a dictionary containing every possible value for the column ordered by frequency, and represents the value as the integer position on this table. Appropriate for low or medium cardinality unordered columns [16].
  - *Bit-Vector Encoding.* In this encoding technique, a bit-string whose size is equal to the table size is associated with every distinct value of the column. Hence, the *i-th* bit of the string x is set to 1 if the value of the *i-th* record is x for this particular column. This is a simplification of *bitmap indexes*, commonly used in row-oriented DBMSs. Due to the recent application of this technique to column-store, much work has been done trying to further compress these bitmaps [17] [18] [19].

- **Late materialization.** Column-store DBMSs usually delay the materialization of records until it is absolutely required to proceed executing the query [20]. In fact, in some cases the DBMS can completely avoid to materialize sets of columns into records by postponing it inside the query plan. However, that means that the entire query execution engine must be specialized and able to execute operations over sets of columns instead of records.

- **Query Execution Over Encoded Data**. Similar to late materialization, column-stores try to delay the process of decoding data until it is explicitly required — commonly before sending data back to the user. Since several arithmetic operations, filters and comparisons can be executed over columns still in the encoded format, postponing decoding can reduce the amount of data to be decoded or even in some cases completely avoid it.

- **Cache Efficiency and Vectorized Processing**. The execution pattern of several operators, such as a filter over a column, for instance, are tight *for* loops that can greatly benefit from CPU caches and pre-fetching, since columns are laid consecutively in memory. In addition, vectorized CPU instructions can be leveraged to

increase performance and execute selections, expressions and other arithmetic operations considering that column values are usually small and can fit inside CPU registers and caches [21].

Explicit vs. Virtual Row ID

Since column-oriented DBMSs store records one column at a time within different column files, there must be a strategy to correlate these values and materialize records again. A naive approach is to associate an explicit *row-id* to each value in a column file, so that full records can be reconstructed (joined) based on these ids. In practice, keeping these ids alongside the data itself bloats the data size stored on disk and reduces I/O efficiency.

A second approach adopted by most modern column-store DBMSs is to infer a *virtual row-id* based on the offset inside a column file. For instance, if the representation used by column A is fixed-width, the *i-th* record offset can be easily expressed as $i * column\_width(A)$. In the case of non fixed-width columns though, more metadata needs to be maintained by the DBMS to find the exact offset of a particular record. One technique is to store the virtual row-id of the first value within a block, or even the row-ids of a few scattered values inside the block so that less data needs to be scanned, given that data is usually encoded and stored in blocks [22].

One restriction of the virtual row-id strategy is that it requires the DBMS to store all columns using the same row-id based ordering. Figure 2.1 illustrates the differences between these two approaches and a row-store.

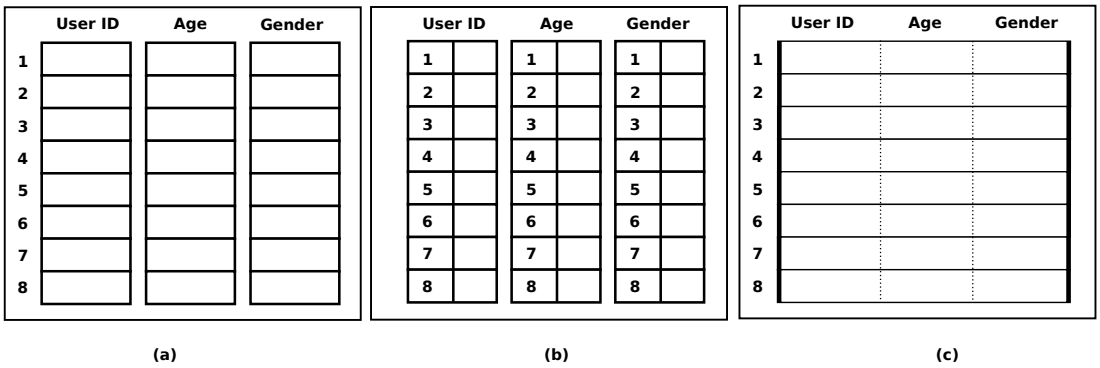

Figure 2.1: Different DBMS architectures data layout. (a) illustrates a column-store using virtual row IDs, (b) shows a column-store with explicit row IDs and (c) illustrates the layout of a row-oriented DBMS.

2.2.1 Column-Store Designs

To the best of our knowledge, at present time there are two basic data organization strategies implemented by most modern column-store DBMSs. A first approach, which we denote as *append model*, appends data to each column file whenever a new record is ingested by the DBS. A second and more recent alternative used by the *C-STORE* [4] architecture consists in keeping the column files sorted according to some pre-defined order. The system can also keep multiple copies of these column files using different sort orders, at the cost of extra storage usage. In both models, columns are sorted using the same order, so that virtual row IDs can be used to reconstruct tuples without incurring in the overhead of storing explicit row-ids.

Append Model

Column-oriented DBMSs using this model append data to the end of column files every time a new row is ingested by the DBS. Because data ingestion usually occur in batches on column-oriented systems instead of one row at a time, data for each column is batched in memory, encoded and written every time it hits some threshold in order to make efficient use of disk I/O. The same assertion holds for in-memory systems to better leverage memory locality and reduce cache misses [23].

This simplistic strategy has the advantage of providing very efficient ingestion ratios since no reorganization is required at load time [24]. However, select operators are com-

putationally expensive and a full data (column) scan is needed unless some kind of index is maintained by the DBMS, considering that the DBMS has no information about which values (or ranges) are stored into a particular block of a column file.

MonetDB [25] [24] is a column-oriented DBMS that adopts this strategy, where the dataset is not stored in any particular order but rather only appended to. MonetDB focuses on executing simple operations over single columns in a very efficient manner by exploiting CPU SIMD operations, tight *for* loops and minimizing cache misses. The entire query engine is based on composable BAT (*Binary Association Table*) algebra, in which operators can consume one or a few BATs and generate new BATs. However, since there is no notion of indexing, entire columns need to be scanned when a particular BAT needs to be evaluated, independently of the filters applied [21].

Column files in MonetDB are memory mapped so that the overhead of reading and writing data to disk is minimized, by the cost of giving the control of IO scheduling to the OS. This also causes MonetDB to lack support for compression and advanced encoding. A few of these shortcomings were addressed by a subsequent rewrite, called VectorWise [26], but the fundamental BAT albegra and indexing characteristics remained the same.

C-STORE

The C-STORE architecture was first proposed by Stonebraker *et al.* in 2005 [4], and later implemented and formalized by Adabi [27] in 2008. Column-oriented DBMSs following the C-STORE architecture store data vertically partitioned, one column per file, order by one or a set of the table's attributes. Select operators applied to the columns on which the table is ordered by can be executed efficiently, similar to what happens with indexes in traditional row-oriented DBMSs. In cases where fast access is required through more than one column, C-STORE can maintain multiple copies of the table ordered in different ways, trading storage space for query efficiency. Each one of the copies of a particular table, also called *projections*, contains one or more of its columns and a sort order defined according to workload requirements. Thereby, at query time the query planner can choose the best projection to use for a particular query; typically,

there always is at least one projection containing all columns of a table able to answer any queries, also called *super projection*.

For example, in a table containing the following attributes:

$$(user\_id, user\_country, user\_age, user\_device)$$

a query that counts the number of distinct users whose age is over a certain threshold in a specific country can benefit from a projection containing $user\_id$, $user\_country$ and $user\_age$, ordered by $(user\_country, user\_age)$.

Since all data in a C-STORE DBMS is stored following a pre-defined order, all projections of a table need to be evaluated and re-ordered every time new records are loaded, which can be a very costly operation in terms of both disk I/O and CPU. The costs of maintaining these ordered copies of the data is amortized by C-STORE by keeping a dual data store model composed of:

- **Write Optimized Store (WOS)**. A in-memory store used to buffer records in a row-oriented and unencoded format in order to speed up the ingestion of new records.
- **Read Optimized Store (ROS)**. Disk based data store used to persist per-column data once it is properly ordered and encoded.

Periodically, a background task called *tuple mover* is executed in order to flush data from WOS to ROS by ordering, encoding and writing each column to a new ROS container (procedure known as a *moveout*). Likewise, the same tuple mover also merges smaller ROS containers into bigger ordered files in case they meet some established criteria (also called *mergeout*), in order to prevent columns to become too fragmented and deteriorate I/O efficiency. Considering that data for a particular column can be stored in several places at any given time, both WOS and all ROS containers need to be scanned at query time. Ultimately, C-STORE's tiered storage model is similar to the one used by *Log Structured Merge Trees* [28].

In order to prevent a full data re-organization when records are deleted, C-STORE has a special column that dictates whether a record has been deleted or not. That also enables the implementation of MVCC based concurrency control and the execution of queries is the near past. Similarly, record updates are handles as a deletion of the original record and insertion of a new one.

2.3 Indexing Methodologies

Considering that row-oriented DBMSs store data on disk record by record usually in the same order as they are ingested (unless clustered indexes are used — more on than in Chapter 3), any select operator would require all records to be evaluated and consequently read from disk in case no other structures are used. If we consider a table containing N records, the cost of any selector operator in a naive row-store DBMS, measured in number of records that need to be touched, is $\mathcal{O}(N)$. Notice that this burdensome process of reading an entire table and materializing every single row — frequently called a *full table scan* — is not flawed by itself, but rather dependent on the amount of records that actually match the select operator.

In order to avoid full table scans and reading unnecessary records, row-oriented DBMSs can keep auxiliary index structures that provide fast access to records. These index structures are usually maintained on a per column basis, in such a way that select operators based on this column can be answered by traversing these structures instead of evaluating every possible record. Historically, most DBMSs vendors leverage some type of *B+Tree* index structure [29] [30] because of its efficiency when working in conjunction with block based storage devices, but other indexes based on hash tables and special bitwise representations such as *bitmap indexes* [31] are also common. Recently, *skip-lists* [32] are also being exploited for in-memory DBMS vendors [33] due to their simplicity and flexibility.

Different index types have particular performance characteristics and are appropriate in different scenarios. For instance, indexes based on hash tables are used for exact match select operators (*WHERE column = value*) providing amortized constant time for lookups, but cannot answer range queries (*WHERE column > value*) [34]. Similarly,

bitmap indexes provide constant lookup time for exact match select operators, but are usually faster in practice than hash tables by avoiding hashing and *re-bucketing*. However, bitmap indexes are only practical for low cardinality columns due to their fast growing size [35]. Lastly, B+Tree indexes are more flexible and thus can support both exact match and range select operators, but are also more intensive to update.

The two biggest drawbacks of maintaining per-column indexes are (a) the overhead of keeping these structures up-to-date and (b) additional resource utilization. The logarithmic overhead associated with updating data structures such as B+Trees, for example, may slow down insertions and updates to the point where DBMS vendors advice users to drop indexes before loading bulks of records, and re-create them after the load is done [36]. In short, choosing when to use a database index as well as the most appropriate type is not an easy task and should be done carefully. In the next Subsections we discuss a few approaches to help automate this task.

2.3.1 Offline Indexing

This first approach that can be used to tackle the index creation problem, or in more generic terms to automate the generation of a physical design, is called *offline indexing*. Offline indexing is a task usually performed by the database administrator (DBA) with the help of a auto-tuning tool, commonly provided by major DBMS vendors [37] [38] [39]. The DBA inputs a representative set of the workload into the auto-tuning tool, that after executing a sequence of *what-if* analysis [40] outputs a few physical design suggestions, such as which indexes to create and which indexes to drop. The suggestions are therefore evaluated by the DBA, who will ultimately decide whether to apply them or not.

Besides still requiring a considerable amount of manual intervention, this approach is also flawed in cases where the workload is not known beforehand or not known in enough details to drive informed physical organization suggestions. In addition, this static strategy also does not take into account that the DBS organization and workloads may evolve with time.

2.3.2 Online Indexing

Online indexing addresses the limitations of offline indexing. Using this approach, instead of making all decisions in advance, the DBS constantly monitors and re-evaluate the current physical design based on observed characteristics of the workload. Through these observations, the DBS periodically recommends changes to the physical design and can create or drop indexes, for example, without DBA intervention [41].

Despite addressing the problem of workloads evolving over time, re-organizing the physical design can be a resource intensive operation and require a significant amount of time to complete. This approach is appropriate to workloads not very dynamic, that slowly change over time. In highly dynamic scenarios such as the ones found in exploratory workloads, this strategy can lead to a situation where by the time the DBS finishes re-organizing itself, a new organization is required because the workload has changed again.

2.3.3 Adaptive Indexing

Adaptive indexing addresses limitations from both online and offline indexing by instantly adapting the physical design to workload changes. Instead of building indexes as background procedures, the DBS reacts to every query issued with lightweight actions, that refine incrementally indexes utilized by the query. The more queries are executed, the more indexes are incremented and better performance is achieved for this particular workload. In such a way, the physical design slowly and adaptively evolves to better accommodate workload changes as they arrive, without incurring unnecessary overhead [42].

Adaptive indexing has been extensively studied in the last decade, both in the context of rows-stores [43] and column-stores; in the latter, an adaptive indexing technique called *database cracking* has become focus of research in the last few years [44] [45] [46]. Database cracking will be further discussed in Subsection 2.3.4. Lastly, a new approach called *holistic indexing* [47] proposes that CPU idle cycles in the system should also be used to refine the indexes in addition to in-query incremental indexing, mainly in cases where the DBS does not execute enough queries to incrementally build its indexes adequately.

2.3.4 Database Cracking

Database cracking [44] is an adaptive indexing technique that re-organizes the physical design as part of queries, instead of part of updates. It has been widely studied in the last few years in the scope of column-store DBMSs [45] [46] [48] [49] [50], but the same ideas can also be applied to row-oriented DBMSs. Using this technique, each query is interpreted as a hint to crack the current physical organization into smaller pieces, whereas the remaining parts not touched by any queries remain unexplored and unindexed.

Once the first query hits the DBS containing a select operator on column A, a new version of this column called *cracker column* (A_{CRK}) is created, in addition to a *cracker index* that maintains information about how the values are distributed in A_{CRK}, usually implemented as an AVL tree. For instance, a query containing the selector operator $A < 30$ breaks A_{CRK} into two unordered pieces, the first one containing all values lower than 30 and the second one comprehending values greater or equal to 30, using a strategy much similar to quicksort [51]. A subsequent query containing the select operator $A < 20$ only needs to scan the first piece of A_{CRK}, breaking it further into two smaller pieces. Therefore, after these two queries A_{CRK} is composed by three unordered pieces: (a) one containing values from 0 to 20, (b) containing value between 20 and 30 and (c) containing the remaining values. The cracker index is responsible for keeping metadata and quickly locating the cracked pieces.

In a cracked DBS, the physical organization is driven by the queries themselves and can adapt if the workload changes over time. However, since the cracked column has a different sort order than the one used by the original table representation, it cannot leverage inferred virtual row IDs — need to explicitly store them — potentially doubling the cracked column's storage footprint. In addition, in highly dynamic DBS it is very likely that for each query (or every few queries) some level of re-organization will be required, increasing disk I/O utilization.

2.4 Database Partitioning

Partitioning is an umbrella term used in DBS to describe a set of techniques used to segment the database — or a particular table —- into smaller and more manageable parts, also called *partitions*. Each partition can therefore be stored, accessed and updated individually, without interfere with the others. Partitioning can also be used to increase performance, for instance, by accessing and evaluating multiple partitions in parallel or to provide high-availability in distributed DBSs by storing replicas of each partition in different nodes. Other uses of partitioning include security and to separate frequently accessed data to data rarely touched.

In a two-dimensional relational world composed of records and attributes, there are two obvious orientations to partition a database: (a) vertically, by splitting columns or sets of columns into separate partitions or (b) horizontally, by assigning different records to different partitions, commonly based on some *partition key*. Column-oriented DBMSs, for example, are systems completely partitioned vertically, in such a way that each column is stored separately in its dedicated partition. On the other hand, most row-oriented DBMSs support some kind of horizontal partitioning [52] [53], and even some column-stores further partition the data horizontally for manageability and performance [54].

A few different criteria can be applied over a partitioning key in a horizontally partitioned table to dictate the mapping between records and partitions: (a) *range partitioning*, that maps records to partitions based on pre-defined ranges; (b) *list partitioning*, by pre-defining a list of values to include in each partition; (c) *hash partitioning*, that uses a hash function to determine the membership of a partition; and (d) *composite partitioning* that uses some combination of the other techniques.

Partitioning can also be leveraged to prune data at query time. A database table partitioned by one column can efficiently execute filter operations over it by completely skipping all partitions that do not store any data that could match the filter being applied. For instance, consider a table partitioned by *col1* and a query containing the exact match filter *col1 = 10*. All partitions that cannot store the value 10 can be safely skipped, independent of the criteria used for partitioning. Certain types of filters, however, cannot

be used to skip data for some partitioning criteria, as is it the case for range filters and hash partitioned tables — unless an *order-preserving* hash function is used [55], what could remove the benefits of even data distribution from hash partitioning.

CHAPTER 3

INDEXING MULTIDIMENSIONAL DATA

Multidimensional data is widely used in several areas of computing, such as geographic information systems (GIS), robotics, computer vision, image processing and OLAP DBSs. According to the most common definition [56], multidimensional data can be considered a collection of points or objects in a k-dimensional space, providing that k (number of dimensions) is greater than one. When the values for each dimension describe one object's physical position in the space — a set of points-of-interest in a geographic system, for instance —, this data set is characterized as *locational*. On the other hand, several applications leverage multidimensional data to represent objects and their different characteristics (or *features*), such as image processing and multidimensional DBSs, which are therefore named *non-locational*.

Aside from what the data itself represents, users and data analysts usually require efficient methods to query, analyze, transform and more generally to manipulate the datasets. In the last few decades, several data structures specially designed to store and index multidimensional data (also known as *multidimensional data structures*) have been proposed, each one of them with theirs own benefits and drawbacks, mostly differing with regards to the following characteristics [57]:

- *Dimensionality*. While locational applications may require only two or three dimensions, certain types of application, like multidimensional DBSs, can require several hundreds of dimensions.

- *On-disk format support*. Even though primary memory has become cheaper such that today it is fairly common to find servers with more than 1TB of memory, some of these data structures are optimized to operate over data stored on disk and minimize the number of pages read and written to disk.

- *Dynamicity.* Some applications can tolerate static data structures pre-computed at creation time, but most use cases require support to insertions and deletions interleaved with queries.

- *Supported operations.* Since different data structures can support a different set of operations, the most appropriate data structure should be chosen considering the range of operations required by the application, *e.g.*, exact searches, range searches and nearest neighbor queries.

- *Time efficiency.* Time complexity of insertions and query operations relative to the number of points stored.

- *Simplicity.* Very complex data structures with multiple corner cases are usually harder to maintain and more error-prone.

An index is a data structure used to improve the efficiency of data retrieval operations over a particular dataset, at the cost of increased storage usage and update overhead. A database index is an index data structures created over one or a few columns of a table, allowing select operators (filters) over these columns to locate data in sub-linear time, or without having to scan all records every time the table is queried. In addition to fast lookup, indexes are also leveraged to police and enforce some database constraints (such as *unique*, *primary* and *foreign keys*), and for certain index types to enable ordered iteration over the stored records. Today, indexes are ubiquitous and present on virtually every commercial database [58] [12] [34] [33] [36].

Indexes can be built on top of numerous data structures. Most disk-based indexes leverage some kind of B-Tree [29] in order to increase block locality and use I/O more efficiently, but as in-memory DBMSs become commonplace other types of balanced trees and even data structures like *skip-lists* [32] are commonly used. Tree based indexes usually provide logarithmic access time, support for range queries and ordered iteration and usually are less susceptible to the input data distribution, being able to remain balanced even in the presence of very skewed inputs.

Other index types such as the ones based on hash maps and bitmaps do not support range queries or ordered access, but can provide faster lookup time. Specifically, hash based indexes provide amortized constant time lookups but can degenerate in certain combinations of input and hashing algorithm. Lastly, multidimensional data structures such as *R-Trees* [59], *Quad-Tree* [60] and *K-D Trees* [61] are also available in a few specialized DBMSs [62] [63].

In the following Sections we describe some of the most popular one-dimensional and multidimensional indexing data structures and discuss their applications on DBSs.

3.1 One-Dimensional Indexes

A one-dimensional index is a data structure that provides fast access to a dataset given the value for a single column (or feature). Usually one column is chosen to be the key based on which the index will be constructed and maintained, and every subsequent query based on the value of this particular column can be answered in sub-linear time.

One-dimensional indexes are usually leveraged to build primary keys within databases, even though the same data structures can be applied for secondary indexes. Hence, solely based on the value of this column, the records that match the search predicate can be found through some kind of algorithmic operation over the index structure. However, any query filtering by other attributes cannot leverage the index and need to scan the entire dataset.

One-dimensional indexes are not necessarily built on top of a single column though; in fact, some one-dimensional index implementations support up to 32 columns as keys [58]. The main difference between these types of indexes and multidimensional indexes is in the type of queries supported. One-dimensional indexes spanning multiple columns based on hash can only execute a fast lookup if exact values for all key columns are given, whereas tree based indexes can execute lookups in sublinear times as long as the values provided form a prefix of the index key. For example, if the index is created over the columns (a, b, c), fast queries can be executed over a, and a, b, but not b or a, c.

Classic one-dimensional indexing structures are usually based on hashing or hierarchi-

cal data structures. Other common strategies are the use of bitmaps and, more recently, skip lists. These data structures are discussed in the following subsections.

3.1.1 B-Trees

B-Trees [29] are self-balanced hierarchical data structures that maintain data sorted and allow ordered iteration as well and search, insertions and deletion in logarithmic time. B-Trees are a generalization of binary search trees, in which a node can hold more than one value and thus have more than one child. B-Trees are specially suited for disk storage considering that each node can contain a large number of values and be stored in a single disk page, minimizing the number of disk pages read from disk in order to execute a lookup if compared to a regular balanced binary tree. The height of a balanced binary search tree is proportional to $log_2(n)$, whereas the height of a B-Tree containing m values per node is proportional to $log_m(n)$, where n is the number of values in the tree. Consequently, as m grows to fill the space of a disk block, the height of the tree decreases thereby reducing the number of disk pages that need to be visited on a lookup.

Each internal node on a B-Tree has an upper and lower bound on the number of values to store. These boundaries prevent the tree from degenerating, what could lead to non-efficient storage utilization and unbalancing. All nodes that drop below the lower bound threshold are removed and its child nodes are distributed among adjacent nodes on the same level. In the same way, nodes that exceed the upper boundary are split in two. Splits might need to propagate up the tree structure.

B+Trees [30] are a variation of B-Trees in which internal nodes only store pointers to child nodes but do not store any values, in such a way that all values on the tree structure are stored on the leaves. In addition, all leaves are linked in order to provide fast ordered iteration without requiring access to any internal node. Another advantage of B+Trees is that internal nodes can usually fit more pointers inside a single disk block, thus reducing the fanout of the tree since they do not store values but rather only pointer to child nodes.

B+Trees are extensively used on disk-based DBMSs to build primary keys and secondary indexes due to the reduced number of disk pages required to be read in order to

execute a lookup. B+Trees can be used in database indexes in two main ways. First, the data can be organized in arbitrary order while the logical order is dictated by the index. In this case, also called *non-clustered indexes*, the B+Tree leaves store pointers to the record location. A second approach is to store the record data inside the leaves themselves. In this index organization, also called *clustered*, the record data is stored in an ordered manner, greatly increasing the sequential read order. Since clustered indexes control the order in which the records are stored, there can only be one clustered index per database table.

3.1.2 Hash Tables

Hash Tables are data structures used to store $key \rightarrow value$ pairs in a sparse manner. All values in a hash table are organized in an array of *buckets*, which are structures able to store those pairs of values, and a hash function is used to map a given key to a target bucket. If the hash table is well sized in a way that each bucket stores only a single value, lookups can be done in constant time. Theoretically, lookup time complexity is also proportional to the maximum key size in order to apply the hash function, but we will assume that the key size is negligible if compared to the number of entries on the table.

In practice, since the number of $key \rightarrow value$ pairs is usually larger than the number of buckets, a single bucket may need to store several values. In order to lookup keys that map to buckets storing multiple values, all entries may need to be scanned and this deteriorates the constant lookup time guarantee. However, assuming that the bucket array is properly sized and hash collisions are infrequent, lookups can be still executed in constant time on average, or in *amortized constant time*.

Hash Collisions

Due to the Birthday Problem (or Birthday Paradox) [64], hash collisions are practically unavoidable as the number of $key \rightarrow value$ pairs to store increases. Therefore, all hash table implementations must have some strategy to handle hash collisions. Some of the

most common resolution strategies for hash collision are:

- **Separate Chaining**. In a hash table with separate chaining, each bucket holds a pointer to a linked list of values. Collisions are resolved by appending the collided values to the appropriate linked list, and lookups and deletes are executed by scanning the entire collision list. Alternatively, other data structures such as binary search trees can be used to stored collided values and provide better search guarantees as the number of values per bucket increases.

- **Open Addressing.** The term *open addressing* refers to the fact that the value location is not only determined by the hash function. When using this approach, every time a collision occurs an alternate unused bucket is allocated; at look up time, multiple buckets may need to be probed until the correct value is found or reaching an empty bucket. Some of the commonly used strategies to determine the probing sequence once a collided element is found are:

  - *Linear Probing.* The probing offset is 1, *i.e.*, buckets are probed sequentially.
  - *Quadratic Probing.* The probing offset increases quadratically. This probe sequence alleviates the effect hash clustering has on linear probing.
  - *Double Hashing.* The probing offset is calculated through the use of a secondary hash function. This probe sequence also improve efficiency in the presence of hash clusters, but it is slower than linear and quadratic probing due to the additional hash computation required.

  The drawback of the open addressing method is that the number of stored values cannot exceed the number of available buckets. In fact, it has been shown that lookups and insertions degrade significantly if the load factor (ratio between number of stored elements and number of buckets) grows beyond 70% [65]. In these cases, some dynamic table resizing table technique must be leveraged.

Re-Hashing

Independent of the hash collision strategy being used, the overall hash table performance deteriorates as the load factor increases. For Separate Chaining the linked lists can grow too large, whereas for Open Addressing the probing steps can ultimately require scanning the entire table. Therefore, most hash table implementations make use of some *re-hashing* procedure, which basically consists in building a new hash table with greater capacity (usually doubling the previous size) and moving all pairs from old to the new table. However, since the capacity of these tables is different, a single copy would not maintain the correct mappings between key and target bucket and thus all elements need to be re-hashed.

The trivial linear strategy of re-hashing each single element can be prohibitively expensive if the keys or values are large or if the cost of moving data between buckets is high. This is usually the case for disk based and distributed systems. For these cases, some *incremental resizing* technique is required. The basic rationale behind incremental resizing is that by incrementally resizing the table on insertions and lookup operations, the overhead of rebuilding the entire table can be amortized. Some of the most important incremental resizing techniques are:

- **Linear Hashing**. Linear Hashing [66] [67] allows incremental hash table expansion and amortizes the computation required to rebuild the entire table through the execution of frequent single bucket expansions. Linear Hashing maintains a collection of hash functions H_i, where the interval range of H_{i+1} is twice the size of H_i. A pointer p is also maintained is such a way that buckets greater than p have already been split (or re-hashed using H_{i+1}) and buckets below p still use H_i. In order to keep a low load factor, the bucket pointed to by p may be split and p incremented. When p reaches the end of the range, i is incremented, *i.e.*, the ranges are doubled.

  The main drawback with Linear Hashing is that buckets split on each step are not necessarily full; in the same way, full buckets are not necessarily split.

- **Extendible Hashing**. In Extendible Hashing [68], each hash is treated as a binary bit string and buckets are stored using a central directory implemented as a Trie

(or Prefix Tree) [69]. When a bucket split is required, the directory doubles in size by including one more bit on it, and all the new pointers are initialized to the corresponding buckets but the one being split, which is divided into two buckets.

- **Consistent Hashing**. Consistent Hashing [8] is a hashing technique that guarantees that only part of the keys need to be moved when the number of buckets changes. All buckets are hashed and placed in the edge of a circle (also called *hash ring*). Keys are associated with buckets by finding the key's position on this circle and circling clockwise around it until encountering a bucket. When a bucket is added to this circle, only the keys between the new bucket's hash and the previous bucket in the circle need to be moved. On overage, assuming the hash function has perfect distribution, K/B keys need to be moved per bucket creation, where K is the number of keys and B is the number of buckets. Consistent Hashing is extensively used for Web Caching [70] and distributed data stores such as Distributed Hash Tables [71] [72].

Hash based indexing is widely used in DBSs. Most traditional DBMSs implement B+Trees and hash based index types [58] [34], which are suited for different use cases. While hash based indexes are usually faster than the ones based on trees (amortized constant time versus logarithmic), hash indexes can only execute exact match lookups, not supporting range filters or ordered iterations.

3.1.3 Bitmaps

Bitmaps or *Bit Arrays* are array data structures that compactly store bits. Bitmaps can be used to efficiently index low cardinality columns from a database table, or the ones comprising only a handful of distinct values [31]. On its basic form, a bitmap index maintains one bitmap per distinct value of the column and contains one bit per table record. Each bit then controls whether a particular record matches the value corresponding to this bitmap or not, and queries can be executed using bitwise logical operations.

Even though conceptually one bitmap is needed per distinct value, there are several

encoding techniques able to reduce the number of bitmaps required to index a column [35] [73]. Using encoding techniques, it is possible to index n distinct values using only $log(n)$ bitmaps [74]. The rationale behind these encoding techniques is that even though one bit from each bitmap is needed to represent the column value from a particular record, in practice only one of them can be set to 1 (true).

Bitmap indexes have significant performance and space advantages if compared to other index types to index low cardinality columns. The drawback is that bitmap indexes are usually less efficient to update if the base table is stored in a particular order, *i.e.* if the table contains a clustered index. In that case, insertions of new records can require a burdensome bitmap re-organization. Therefore, they are commonly used over mostly static datasets and generally unsuitable for OLTP workloads.

Much work has been done towards compressing these bitmaps in order to save space. Most compression schemes use some variation of Run-Length encoding [15], being the most common: *Byte-aligned Bitmap Code* (BBC) [75], *Word-Aligned Hybrid Code* (WAH) [76], *Partitioned Word-Aligned Hybrid* (PWAH) [77], *Position List Word Aligned Hybrid* [78], *Compressed Adaptive Index* (COMPAX) [79] and *Compressed and Composable Integer Set.* (CONCISE) [80]. Ultimately, the efficiency of most of these compression techniques also depends on sorting the base dataset, what can reduce the index size by a factor of 9 in some cases and greatly improve lookup speed [81].

3.1.4 Skip Lists

Skip Lists [32] are data structures that provide logarithmic lookup time over an ordered sequence of elements. Each element on a skip list can contain one or more pointers. The first pointer always points to the next element on the list whilst subsequent pointers skip an increasing number of elements. Lookups on these structures are executed by iterating over the more sparse level of the list until two consecutive elements are found, one smaller than the target value and one larger or equal. If the second element is equal to the element being searched for, the lookup is done. If the second element is larger, the procedure is repeated using the next level (lower) in the pointer hierarchy.

The number of pointers maintained by one list element is called its *height*. When a new element is inserted into a skip list, its height is determined probabilistically by tossing a coin successive times. Hence, an element of height n occurs once every 2^n times. Once the element's height is determined, it is linked in every level to the other lists. On the average case, lookups, insertions and deletions can be executed in logarithmic time, but since skip lists are probabilistic data structures that can degenerate, the worst case is still linear.

B+Trees are a better fit for disk based DBSs since they minimize the number of pages required to be read in order to answer a query, but skip lists are becoming popular among memory-only systems. MemSQL [33] is the first in-memory DBMS to leverage skip lists as its primary database index. The main advantage of skip lists is its simplicity, being more concise and simpler to implement if compared to B+Trees. Moreover, complex operations such as page splitting and rebalancing and not required.

3.2 Multidimensional Indexes

Multidimensional indexes are based on the notion that more than one column should constitute the primary key and that all columns in the key should play an equal role on determining record placement. In addition, the index is required to support operations such as the retrieval of all records that match a partially specified primary key. For example, a three-dimensional index data structure needs to be able to retrieve all records matching values specified for any two columns but one, values specified for only one column but none for the other two, or even combinations of ranges and exact match values or ranges for all three dimensions.

The alternative of building one one-dimensional index per column is insufficient since the traversal of each structure needs to be independent, by generating one list of records matching each index and calculating their intersection. This is a costly operation and it does not leverage the potential restrictivity of one column in the traversal of the other indexes. Moreover, a second alternative of building a single one-dimensional containing all column is also unsatisfactory since values for all three columns need to be specified

in order to execute a lookup, and range queries can require a full scan depending on the column order and the index type being used.

The following subsections describe the main data structures used to index multidimensional data and their trade-offs.

3.2.1 Multidimensional Arrays

Perhaps the more natural data structure to store multidimensional data are multidimensional arrays [82]. Multidimensional arrays provide fast (constant time) access to any element stored, since they can be easily derived from the dimension indexes by using a row-major numbering function. However, these data structures usually suffer from two main issues when storing large datasets in high-dimensions: (a) in order to quickly locate a particular element in the array, the number of values in a dimension as well as the number of dimensions themselves must be static, meaning that any changes require a re-organization in the entire structure; (b) the number of elements grows exponentially as the number of dimension increases, what generally creates very sparse data structures.

The use of extendible arrays [83] [84], that can overcome memory managers fragmentation and grow or shrink on-demand may help addressing (a), but sparsity is still a problem that only gets worse as the data set grows bigger. A data compression technique called *chunk-offset* [85] can be used to decrease the memory use of sparse arrays and alleviate (b) by compressing repeated values into a tuple containing $(value, number\_of\_appearances)$, at the cost of requiring a binary search on queries, deteriorating the constant time random access time guarantee.

3.2.2 Grid Files

Grid files are multidimensional data structures designed to store point data in a disk optimized format [86]. In a grid file, the space is segmented using a *grid*, i.e., a set of d-dimensional hyperplanes that divide each dimension of the search space in a non-periodic fashion. Every cell in the grid is called a *bucket* and usually store a small set of points that can be stored within a single disk page.

In order to guarantee that queries require no more than two disk accesses to execute, the grid file itself is represented by d one-dimensional arrays, also called *scales*, that point to the on-disk location of the point buckets. The scales are also stored on a disk page or on consecutive disk pages, depending on its size. Even though each grid cell contains a reference to exactly one bucket, a bucket may represent several neighbor cells when they do not hold enough points to fill an entire bucket. This optimization helps to keep low the number of disk pages used.

When executing search queries, a first disk access is required to read the grid file from disk — it can be avoided if the structure is kept in memory — and after finding the target bucket, one more disk access to read it. To insert a new point, a search for the exact data point is first conducted to locate the cell in which it should be stored. If the bucket associated with this cell is not empty, the point is simply inserted in this bucket. Otherwise, one needs to check if any of the current hyperplanes can be used to split this bucket; if none of them are suitable, a new hyperplane is created to divide this bucket and the scale is updated.

3.2.3 Tree-based Data Structures

One alternative to multidimensional arrays is the use of tree based data structures for multidimensional data. Tree based data structures are commonly used to store polygons and other geometric shapes, but they can also be adapted and optimized to store points. In the following Subsections we describe some of tree based multidimensional data structures most relevant to this work.

3.2.3.1 K-D-Trees

A K-D-tree store multidimensional data points in a binary tree [61]. Each internal node of this tree stores one data point and divides the space into two using a d-1 dimensional hyperplane that contains the current data point. The two children node are pointers to a subtree that represent each subspace. In each level of the tree the space is split based on a different dimension, chosen alternately. Searching and inserting operations are a

simple matter of traversing the tree using the correct subspaces. Deletions are somehow more complicated operations and may require the reorganization of the subtrees below the removed node [87].

The biggest drawbacks of the original K-D-tree are: (a) the data structure is not balanced, meaning that the organization of the tree depends on the order that data points have been inserted, and (b) data points are scattered all over the data structure what could make it potentially more difficult to implement a on-disk backend store. With those problems in mind, adaptive K-D-trees [88] do not split the subspace using a data point, but rather find a hyperplane that divides evenly the data points into two subspaces. Thus, data points are only stored on leaves while internal nodes only contain the hyperplanes (metadata) that divides the search space.

The problem with this approach is that for some data sets there could be no hyperplane that divides the data points evenly in a given orientation (based on the node level) [89]. In order to overcome this limitation, BSP-trees [90] [91] decompose the space in a binary fashion using a d-1 dimensional hyperplane just like regular K-D-trees, but instead of choosing the orientation based on the level of the tree, the choice is made based on the distribution of the data points. Despite adapting better to different data distribution than regular K-D-trees and adaptive K-D-trees, BPS-trees are still not a balances data structure, and can contain deep subtrees depending on the distribution of the data set.

3.2.3.2 Quadtrees

Quadtrees are tree shaped data structures in which each point inserted on the tree breaks the search space along all d dimensions [60]. Therefore, each level of the tree contains 2^d children pointing to each subspace created by the subdivisions. For a 2 dimensional tree, for instance, each level of the tree divides the space into four quadrants (explaining the name *quadtree*), usually called NE, NW, SE and SW (northeast, northwest, southeast and southwest) — reason why 3 dimensional quadtrees are often known as *octrees*. Even though the quadtrees space division can be generalized and applied to more dimensions, it is usually not a good fit for problems with high dimensionality since the number if

children of each node grows exponentially with the number of dimensions.

One very popular quadtree variation is the *region quadtree* that instead of using input points to segment the search space, divides it in regular sized quadrants and stores them only on leaves, hence facilitating searches and providing better balancing if the data sets are equally spread through the search space. In spite of that, neither quadtrees nor regular quadtrees are balanced data structures [92].

3.2.3.3 K-D-B-Tree

A K-D-B-tree [93] is a hierarchical multidimensional data structure that combines techniques used in both K-D-trees and B-trees [29]. Similar to adaptive K-D-trees, each internal node represents a region of the search space that encloses its children, while leaf nodes store data that is contained in its particular region. All the regions represented by nodes on the same tree-level are disjoint, and their union is the entire search space.

Likewise B-Trees, K-D-B-trees can store multiple entries in both internal and leaf nodes, while ensuring that the data structure will remain perfectly balanced independently of the distribution of the data set. However, since splits may need to be propagated down to the leaves, in addition to up to the root, K-D-B-trees do not guarantee minimum space utilization within nodes, which could potentially waste disk capacity in a disk backed tree.

In order to execute a search operation, the searched element is compared to all children regions in every internal node, following the appropriate path that contains it. When a leaf is found, every data point is then scanned sequentially searching for a match. In order to insert a new data point into the tree, first a search to find the appropriate leaf that should store the new data point is conducted. If the leaf is not full, the point is inserted on it; otherwise, the leaf is split in half by a hyperplane (there are different algorithms and heuristics to estimate and evaluate different splits without incurring high computational costs) and a new region is created on its parent. If the parent is also full, it must be split by a hyperplane too and propagate the new region up to the tree. Since this is hyperplane may also impact lower levels of the tree, this split also needs to be propagated to the leaves.

3.2.3.4 R-Trees

R-Trees are multidimensional data structures designed by Guttman [59], in 1984, able to store different multidimensional objects, including points and polygons. The key idea behind this data structure is to group nearby objects in each level of the tree by using their *Minimum Bounding Rectangles* (MBR). Similar to B-trees, R-trees are balanced data structures and since multiple entries can be stored in a single node, they are also optimized for on-disk storage.

Searching in a R-Tree is straightforward and very similar to B-Trees. At each level of the tree, all MBRs are tested for overlap with the searched object; in case there is intersection, the recursion continues through that subtree. Differently from B-Trees though, multiple MBRs in the same node can also overlap and the search may need to follow *all* subtrees that intersect with the searched object. The more overlap there is between sibling MBRs, the more inefficient queries are.

In order to insert a new object, a single path of the tree must be traversed until finding a leaf, picking the MBR that needs the smallest enlargement to resume the recursion when none of the current MBRs encompasses completely the inserted object. When the recursion gets to a leaf, the object is inserted given that the leaf is not full, updating the corresponding MBR; otherwise the leaf is split into two, both MBRs are calculated accordingly and the split is propagated to the root. Notice that splits only need to be propagated up to the root, never down to the leaves.

Since the split algorithm will ultimately define how good the tree organization is and also have a strong impact on search efficiency, Guttman describe different split strategies in his paper. The most popular R-Tree split algorithm, called *QuadratcSplit*, first searches for the pair of objects that are further apart, and then assign the remaining objects to the set that needs the least area enlargement.

The biggest challenge when building an R-tree is minimizing the overlap between internal nodes (total area covered by more than one node) and coverage (sum of the area covered by all nodes in order to prune more efficiently). With this intention, several modifications to the original R-Tree were proposed in the following years. The R+Tree [94]

avoids overlap of internal nodes by inserting objects in more than one leaf if necessary; therefore, only one path must be traversed on searches, but the tree structure can be larger. R*-Trees [95] attempt to reduce node overlap and coverage using a combination of a reinsertion strategy for some objects in overflowed nodes and a revised split algorithm. Since R-Trees are very sensible to the order data is inserted, deleting and re-inserting some object may result in a better shaped structure. Lastly, R*-Tree's split heuristics are based on minimizing perimeter instead of area, what results in more rectangular areas, and as showed in [95] that reduces considerably the overlap between nodes in their experiments.

Finally, the X-tree [96] is another R-Tree variation that emphasizes the overlap prevention by creating the notion of *super nodes*. Every time the split algorithm cannot find a split that result in no overlap, a super node — that can contain multiple pages — is created. This strategy can improve search efficiency since it prevents overlap between nodes, but in the worst-case scenario the tree can become linearized depending on the workload.

3.2.3.5 PATRICIA-Tries

PATRICIA-Tries [97], or *Practical Algorithm To Retrieve Information Coded in Alphanumeric*, are multidimensional indexing structures based on prefix sharing. In a PATRICIA-Trie, every node represents one bit in the search key, therefore containing at most two children. These Tries are not balanced since its organization depends on the input data distribution, but the degree in which it can degenerate is constrained by the maximum height of the tree, which is equal to the number of bits in the key.

In order to store multidimensional datasets and allow multidimensional keys, the bits representing the values of every dimension can be interleaved in order to construct a single bit string. Hence, every level of a PATRICIA-Trie splits the search space of a specific dimension in half. In [98] this idea is extended by allowing keys of variable length as well as numeric data, and also presenting an approximate range query algorithm that reduces the number of visited nodes by up to 50%.

The PH-Tree [99] is an extension of PATRICIA-Tries in which each level partitions

the space by all dimensions in a similar fashion as Quad-Trees, instead of the binary partitioning used by regular PATRICIA-Trie. This approach reduces the number of nodes in the tree since each level can have 2^k children per node and makes the access independent of the order in which dimensions are stored. However, neither PATRICIA-Tries nor PH-Trees are optimized to index highly dimensional datasets. Lookups on regular PATRICIA-Tries can get close to linear if the dimensionality is large and filters are only applied over a few dimensions – since for all levels where other dimensions are split both paths need to be followed. Moreover, the Quad-Tree node split approach used by PH-Trees can pose substantial memory overhead and the dimensionality scales.

CHAPTER 4

GRANULAR PARTITIONING

Store and index highly dynamic datasets is a challenge for any DBS due to the costs associated with index updates. Indexing OLAP DBSs, in particular, is even more demanding considering that the number of records is substantially higher than in traditional transactional systems. In addition, due to de-normalization the number of columns is larger than in normalized OLTP systems, in such a way that deciding which columns to index in the presence of ad-hoc queries is also challenging. This situation is usually alleviated by the fact that most OLAP systems work over mostly static datasets where load requests are infrequent and done in batches, thus amortizing the index update costs.

However, in order to minimize the latency between data production and the time when the database is indexed and ready to answer queries over the newly ingested dataset, new generation OLAP systems need to be able to continuously ingest data directly from realtime streams. In these realtime scenarios where a large number of load requests are constantly queued up in the system, even small interruptions in serving these requests in order to update indexing structures might stall entire pipelines. Besides delaying data delivery through user queries, these stalls also result in extra resource consumption to process stale load requests in order to catch up with current data.

In this chapter, we describe a new data organization technique called *Granular Partitioning* that extends traditional table partitioning [100] and can be used to efficiently index highly dynamic column-oriented data. The rationale is that by creating more granular (smaller) partitions that segment the dataset on every possible dimension, one can use these smaller containers to skip data at query time without requiring updates to any other data structures when new data arrives. In addition, by properly numbering these containers it is possible to efficiently locate the target container given a record, as well as infer the container offsets based solely on its id, therefore keeping the metadata overhead

low.

The remainder of this chapter is organized as follows. We start in Section 4.1 by describing the target workload and highlighting the characteristics that make it different from traditional OLAP workloads. In Section 4.2 we emphasize why current techniques are not appropriated to index data from this workload. In Section 4.3 we detail a new approach for indexing highly dynamic datasets and in Section 4.4 we present a model and compare the new proposed approach to other database indexing techniques. Finally, Section 4.5 points to related work.

4.1 Workload

In this work, we focus on a type of OLAP workload we refer to as *Realtime Interactive Stupid Analytics*. To the best of our knowledge, this workload differs from traditional OLAP workloads commonly found in the literature by having all of the four following characteristics:

- **OLAP**. Like in any other OLAP ecosystem, this workload is characterized by the low volume of transactions and high amount of data scanned per query which are used as the input to some aggregation function. The dataset is always multidimensional and completely de-normalized.

- **Stupidity**. The term *Stupid Analytics* was introduced by Abadi and Stonebraker [101] to refer to traditional Business Intelligence workloads, as opposed to Advanced Analytics that encompasses more elaborate analysis such as data mining, machine learning and statistical analysis and targets Data Scientists as end users. Stupid Analytics focus on the execution of simple analytic operations such as sums, counts, averages and percentiles over large amounts of data, targeting business users, reporting and dashboard generation. For these use cases, the biggest challenges are in the scale and latency, rather than in the complexity of the analysis itself.

- **Interactivity**. Queries are initiated from a UI where users are able to interactively manipulate the dataset, commonly for reporting and data exploration purposes.

For these user-facing tools, in order to provide a truly interactive experience queries must execute in hundreds of milliseconds [1] up to a few seconds in the worst case, in order not to lose the end users' attention and not to turn the interactions with the dataset painfully asynchronous.

- **Realtime**. In order to generate timely insights, Data Analysts want the ability to analyze datasets as soon as possible, ideally only a few seconds after they have been generated. Since the raw data is usually generated by web logs and transmitted using a log message system [102], the DBS needs to be able to continuously ingest these streams of data with relatively low overhead, in order the maintain the normal query SLA. Moreover, after ingestion the data needs to be immediately available for queries, without being delayed by batching techniques or any other database re-organization procedure.

Apart from these four workload aspects, another characteristic that heavily influences the DBMS architecture to be leveraged is the scale at which Facebook operates. It is common at Facebook to find use cases where people wish to analyze hundreds of billions of rows interactively, in datasets that generate a few million new records per second. Fortunately, there are a few assumptions we can make about the data and query behavior of these workloads that make them easier to handle at scale. The following two subsections describe the assumptions we made regarding the data and queries from the target workload.

4.1.1 Data Assumptions

We start by assuming that all datasets are strictly multidimensional and fully de-normalized. If the base tables are not de-normalized we assume that some external system will be responsible for joining all required tables in either realtime streams or offline pipelines and generate a single input table. Due to the heavy de-normalization these tables end up being wide and containing hundreds of columns.

In addition, we also further simplify the multidimensional model as defined by Kimball

[2] by maintaining the abstractions of cubes, dimensions and metrics, but dropping the concepts of dimensional hierarchies, levels and attributes. Through the investigation of a few of our datasets we have concluded that hierarchies are not extensively used, with the exception of time and date dimensions (for which we can have specialized implementations) and for country/region and city. For the latter example, we can create separate dimensions to handle each hierarchy level and provide a similar data model, still allowing users to drill-down the hierarchy. That considerably simplifies the data model, with the following drawbacks: (a) explosion in the cube cardinality and (b) potentially storing invalid hierarchies (California can appear under Brazil, for instance). However, (a) can be easily overcome by a sparse implementation and we decided to handle (b) externally by using periodic data quality checks. Moreover, we assume that additional dimension attributes are stored elsewhere and looked up by the client tool.

Another characteristic that allows us simplifying the query engine is the fact that the datasets are always composed of primitive integer types (or floats) and *labels*, but never open text fields or binary types. The only string types allowed are the ones used to better describe dimensions (and thus have a fixed cardinality, as described in Subsection 5.2) such as *Male*, *Female* labels for gender dimensions or *US*, *Mexico* and *Brazil* for country dimensions. Therefore, all labels can be efficiently dictionary encoded [16] and stored as regular integers, enabling us to build a leaner and much simpler DBMS architecture only able to operate over fixed size integers.

4.1.2 Query Assumptions

Considering that the dataset is always de-normalized, queries are simple and only composed of filters, groupings and aggregations. The lack of need for joins, subselects and nested query execution in general that comes from the de-normalization enables us to remove a lot of complexity from the query execution engine — in fact, we argue that it removes the need for a query optimizer since there is always only one possible plan (more on that in Section 5.4).

The types or filters used are predominantly exact match, but range filters are also

required (in order to support specific date, time or age ranges, for instance). That means that indexing techniques solely based on hashing, as explained in Section 3.1.2, are insufficient. Support for *and*, *or*, *in* and more elaborate logical expressions for filtering is also required. The aggregation functions required by our use cases are mostly composed of simple functions such as *sum()*, *count()*, *max/min()*, and *average()*, but some projects also require harder to compute functions such as *topK()*, *count_distinct()* and *percentile()*. At last, the challenge of our use cases are usually due to the scale and dataset sizes rather than the complexity of queries themselves.

4.2 Problems With Traditional Approaches

Traditional database techniques used to improve select operator (filters) performance by skipping unnecessary data are either based on maintaining indexes (auxiliary data structures) or pre-sorting the dataset [103]. Maintaining auxiliary indexes such as B+Trees to speedup access to particular records is a well-known technique implemented by most DBMSs and leveraged by virtually every OLTP database. Despite being widely adopted by OLTP databases, the logarithmic cost of maintaining indexes updated is usually prohibitive in OLAP workloads as table size and ingestion rates scale. Most types of indexes (notably secondary indexes) also incur in storage footprint increase to store intermediate nodes and pointers to the data, in such a way that creating one index per column may double the storage usage. Moreover, correctly deciding which columns to index is challenging in the presence of ad-hoc queries.

A second approach to efficiently skip data at query time is pre-sorting the dataset. Column-oriented databases based on the C-STORE architecture [4] maintain multiple copies of the dataset ordered by different keys — also called *projections* — that can be leveraged to efficiently execute select operators over the columns in the sort key. Even though a structure similar to a LSM (*Log Structured Merge-Tree*) [28] is used to amortize the cost of insertions, a large amount of data re-organization is required to keep all projections updated as the ingestion rate scales. Besides, one has to decide beforehand which projections to create and their corresponding sort keys, what can be difficult to

define on datasets composed of ad-hoc queries. Ultimately, since every new projection is a copy of the entire dataset, this approach is not appropriate for memory based databases where the system tries to fit as much of the dataset in memory as possible to avoid costly disk accesses.

Database cracking [44] is an adaptive indexing technique that re-organizes the physical design as part of queries, instead of part of updates. On a database leveraging this technique, each query is interpreted as a hint to crack the current physical organization into smaller pieces, whereas the data not touched by any queries remain unexplored and unindexed. This technique has the advantage of adapting the physical design according to changes in the workload and without human intervention, but has a similar problem to indexing and sorting in highly dynamic workloads — if the ingestion rate is high it might never re-organize itself enough to efficiently skip data at query time.

Lastly, traditional partitioning is usually defined on a per-table basis at creation time by specifying one or a few columns on which to segment a table as well as the criteria to use: ranges, hash, lists of values, intervals or even combinations of these methods are usually supported. Unfortunately, most traditional databases have a low limit on the total number of partitions allowed per table – usually about a few thousands [58] [34] — which prevents its sole use for more fine grained data pruning. Moreover, database systems usually store partitions in different storage containers, in several cases implemented over files, which is not an efficient method to store a large number of containers due to metadata overhead, fragmentation, and directory indirections.

4.3 A New Strategy

In this work, we propose a new technique that extends traditional database partitioning and can be leveraged to index highly dynamic multidimensional datasets. We start by assuming that all tables (or cubes) are composed of columns that are either dimensions or metrics, and that all cubes are range partitioned by all dimensional columns. This partitioning strategy can conceptually be represented by a *grid* that divides the hypercube representation of the dataset in every dimension using periodic hyperplanes, resulting in

equal-sized smaller hypercubes. This property will allow us to simplify multiple operations and make important assumptions about data placement, which will be discussed in the following subsections.

We also assume that the cardinality and range size of each dimension are known or can be estimated beforehand, and that all string values are dictionary encoded and internally represented as monotonically increasing integers. In this way, we can guarantee that the values stored for each dimension are always inside the interval $[0, cardinality[$. Finally, the metric columns are the data stored inside these partitions.

Estimating cardinality and range size might seen tricky at first, but considering that all datasets in our target workloads are highly curated, the expected cardinality can be easily found by querying historical data. Determining the range size is a trade-off between fragmentation and filter efficiency though. If the ranges are too small data can be scattered between several small partitions, but filters will be applied efficiently by skipping all partitions completely outside all query filters. Differently, if the ranges are large, data can be concentrated within less partitions and filters might intersect more containers. We describe how filters are evaluated in Subsection 4.3.3.

Considering that tables can have hundreds of dimensions, and depending on the cardinality and range size defined for each one of them, the database system needs to be able to efficiently handle tens to hundreds of millions of small partitions and therefore exceed the thresholds adopted by most traditional databases [34] [58] [54]. As the number of active partitions scale, a few problems arise: (a) how to represent and quickly find the correct partition for record insertions, (b) how to find the partitions to scan given the filters of a particular query and (c) how to efficiently store millions of small partitions. These items are discussed in the next subsections.

4.3.1 Multidimensional Organization

Given that the multidimensional dataset is organized into partitions that have fixed size in every dimension and the number of segments per dimension is also known (since dimensions have fixed cardinality), storing and indexing these partitions becomes a more

tractable problem. Instead of leveraging memory and CPU wasteful multidimensional data structures based on trees (such as R-Trees, K-D-Treees or QuadTrees), we handle the dataset as being a simple multidimensional array, where each position corresponds to a partition instead of a single data point. Thus, every possible value on a dimension corresponds to one range.

The advantage of this approach relies on the fact the by using a row-major function to assign ids to partitions, the target partition id for a given record can be easily calculated in near constant time (or linear time on the number of dimensions). Furthermore, since the datasets in our workloads are mostly sparse and thus most partitions would be composed of zero records, a hash table can be efficiently leveraged to only keep track of active partitions (the ones with at least one record) and allow amortized constant time lookup based on a partition id, assuming that all partitions are stored in memory. Moreover, due to the sparsity characteristic of the dataset, data inside partitions is also stored in a sparse manner.

However, using this strategy there is no trivial algorithm able to find all active partitions that match a particular set of filters in sub-linear time on the number of active partitions.

4.3.2 Locating Target Partition

The first step required in order to ingest a new record is finding the partition to which it should be appended, based on the values of the dimension columns. Since a row-major numbering function is used to uniquely assign ids to every possible partition, the target partition id can be calculated in linear time on the number of dimensional columns, as well as the opposite direction, *i.e.*, calculating the dimension values (offsets) based on the id.

In theory, several other functions could be leveraged to map the multidimensional partition map into a one dimensional space (the partition id), such as Z-ordering [104], space filling curves [105] [106] and Hilbert curves [107]. All these aforementioned functions add complexity to the id calculation though, with the advantage of improving the locality

of consecutive data points. In our case, however, since the data points are containers comprising multiple records, the speed up achieved by placing different containers closely together in-memory is negligible, according to Amdahl's Law [108]. In fact, the overhead of these calculations usually overcome the benefits and much better results are obtained by guaranteeing that records inside a single container are stored consecutively in-memory.

Using a row-major function, the partition id (*pid*) associated to a particular record d containing n dimension columns, where $\langle d_1, d_2, ..., d_n \rangle$ is the set of dictionary encoded values for each dimension (coordinates) and the cardinality and range size of each column are represented by $\langle D_1, D_2, ..., D_n \rangle$ and $\langle R_1, R_2, ..., R_n \rangle$ respectively, can be calculated through the following equation:

$$pid = \sum_{k=1}^{n} \left(\prod_{l=1}^{k-1} \frac{D_l}{R_l} \right) \frac{d_k}{R_k} \tag{4.1}$$

Notice that even though there are two nested linear functions, the internal product can be accumulated and calculated in conjunction to the external loop in a single pass.

In order to provide constant (amortized) lookup time to the in-memory representation of a particular partition given a *pid*, a hash table that associates *pid* to the in-memory object is also maintained. Thus, at insertion time the system first calculates the target *pid* through Equation 1 and then executes a lookup on this hash table to find the target object. Once the object is found or created (in case this is the first record assigned to this partition), the record is appended to the end of the data structures that store data for the partition, without requiring any type of re-organization.

Region	Gender	Likes	Comments
CA	Male	1425	905
CA	Female	1065	871
MA	Male	948	802
CO	Unknown	1183	1053
NY	Female	1466	1210

Table 4.1: Two-dimensional example dataset.

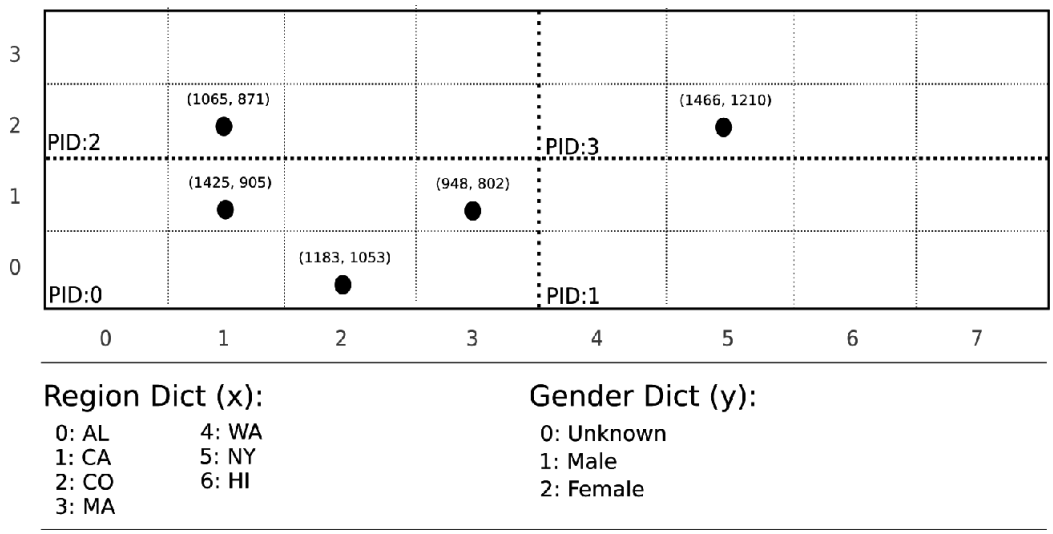

Figure 4.1: Granular Partitioning data layout. Illustration of how records are associated with partitions on the two-dimensional example shown in Table 4.1 and per dimension dictionary encodings.

Figure 4.1 illustrates how data is distributed between partitions given the example two dimensional input dataset shown in Table 4.1. In this example, *Region* and *Gender* are dimensions and *Likes* and *Comments* are metrics. Region and Gender values are dictionary encoded and represented by axis x and y, respectively. Expected cardinality for Gender is 4 and range size 2, while for Region the expected cardinality is 8 and range size is 4. Based on this schema, input records 1, 3 and 4 are assigned to partition 0, record 2 is assigned to partition 2 and records 5 is assigned to partition 3. Partition 1 does not contain any records so it is not represented in memory.

4.3.3 Evaluating Filters

Since the dataset is partitioned by every dimension column, a multidimensional indexing data structure would be required in order to provide sublinear searches by the partitions that match a particular set of filters. However, the logarithmic insertion cost of most multidimensional structures becomes prohibitive when the ingestion ratio scale to a few million records per second. In addition, most multidimensional indexing structures are known to degenerate in high dimensionalities [59] [94] [93].

In our use case, we opted to test all active partitions for every query considering the following reasons: (a) the number of partitions is usually several orders of magnitude lower

than the number of records, (b) the majority of partitions are sparse due to data distribution characteristics, (c) testing every active partition can be efficiently implemented considering that all active *pid*s can be stored in sequential vectors and take advantage of memory locality and cache friendly iterations, and (d) testing partitions against filters is easily parallelizable and chunks containing subsets of the active partitions can be tested by different threads.

Since each server can now be required to store millions of partitions in memory, the following equation can be used to infer the offset on dimension d solely based on a *pid*, and thus considerably reduce the amount of metadata maintained per partition:

$$offset_d(pid) = pid \times \left(\prod_{k=1}^{d} \frac{D_k}{R_k} \right)^{-1} \bmod \frac{D_d}{R_d} \tag{4.2}$$

Therefore, the range of values a partition *pid* stores on dimension d can be represented by:

$$Range_d(pid) = [offset_d(pid), offset_d(pid) + R_d[$$

Based on Equation 4.2, it is possible to infer the offset and range of a partition in every dimension in linear time on the number of dimensions, and at the same time store only a single 64 bit integer per partition of indexing metadata. Also, this equation can be used to evaluate whether a partition matches a set of filters. A filter f_d over dimension d is comprised by a range of values and can be represented by the following interval:

$$f_d = [min, max[$$

Without loss of generality, an exact match filter uses the same representation by assuming that $max = min + 1$. In order to evaluate a filter f_d over dimension d against

partition *pid*, there are three possible scenarios:

- If f_d covers $Range_d(pid)$ completely, the filter can be discarded and the entire partition is considered and scanned for the query.
- If there is no intersection between f_d and $Range_d(pid)$, then f_d is discarded and partition *pid* is skipped for this query.
- If f_d intersects $Range_d(pid)$ partially, then f_d needs to be pushed down and applied against every single record within the partition. Every time a filter intersects a partition, rows that do not belong to the query scope are scanned, reducing pruning efficiency.

4.3.4 Partition Storage

As more columns are included in the partitioning key and consequently the partition sizes decrease, the number of partitions to store grows large. Since the dataset is too fragmented into small partitions, a disk based database is not appropriate for this type of data organization and an in-memory database engine is required. Regarding the in-memory data organization, each column within a partition is stored in a dynamic vector, which provides fast iteration by storing data contiguously and re-allocates itself every time the buffer exhausts its capacity. Data for each of these vectors is written periodically and asynchronously to a disk-based key value store capable of efficiently storing and indexing millions of key value pairs (such as RocksDB [109]) to provide disaster recovery capabilities, and persistency can be achieved through replication.

4.4 Comparison

Table 4.2 presents a model that can be used to compare the worst-case complexity and trade-offs between Granular Partitioning and traditional indexing techniques. This model assumes that the dataset is composed of a single table containing C_T columns and N_T records and that each query can apply filters on any of these columns, whereas C_Q is the

number of columns and N_Q the minimum number of records required to be scanned in order to answer it.

Following this model, the complexity of the query problem should range from $\mathcal{O}(C_T * N_T)$ or a full table scan of all columns in the worst case, to $\Omega(C_Q * N_Q)$ by only reading the exact columns and records required to compute the query and without touching anything outside the query scope. Other aspects that need to be evaluated are insertion costs, or the costs associated with updating the indexing data structures every time a new record is ingested, and space complexity, or the space required to maintain the indexing structures.

	Query	**Insertion**	**Space**
Naive	$C_T * N_T$	C_T	$C_T * N_T$
Row-store (B+Tree Indexes)	$C_T * N_Q$	$C_T * log(N_T)$	$2 * C_T * N_T$
Column-store (C-STORE)	$C_Q * N_Q$	$C_T * log(N_T)$	${C_T}^2 * N_T$
Granular Patitioning	$C_Q * N_Q * \delta$	C_T	$C_T * N_T$

Table 4.2: Comparison between different indexing methods.

We present and compare four hypothetical databases implementing different indexing techniques using our model: (a) a *naive* row-store database that does not have any indexing capabilities, but rather only appends new records to the end of its storage container, (b) a row-store database maintaining one B+Tree based index per column, (c) a column-store database based on the C-STORE architecture that keeps one projection ordered by each column, and (d) a database implementing the Granular Partitioning technique presented in Section 4.3.

Since the naive database (a) does not have any indexing capabilities, every query needs to execute a full table scan independently of the filters applied, and considering that records are stored row-wise, the entire table needs to be read and materialized, *i.e.*, $C_T * N_T$. Nevertheless, insertions are done in C_T since each column only needs to be appended to the end of the list of stored records. Considering that no other data structures need to be maintained, the space used is proportional to $C_T * N_T$.

As for a row-store database maintaining one index per column (b), queries can be executed by leveraging one of the indexes and only touching the records that match

the filter (N_Q), but still all columns of these records need to be read. Hence, overall query complexity is $C_T * N_Q$. Insertions of new records need to update one index per column, so considering that index updates are logarithmic operations the overall insertion complexity is $C_T * log(N_T)$. Furthermore, assuming that the space overhead of indexes is commonly proportional to the size of the column they are indexing, the space complexity is proportional to $2 * C_T * N_T$.

A column-store database implementing the C-STORE architecture and maintaining one projection ordered by each column (c) can pick the correct projection to use given a specific filter and read only records that match it (N_Q). In addition, the database can only materialize the columns the query requires, resulting in overall query complexity proportional to $C_Q * N_Q$. However, insertions require all projection to be updated. Considering that projections can be updated in logarithmic time, the insertion time is proportional to $C_T * log(N_T)$. Moreover, since each projection is a copy of the entire table, the space complexity is ${C_T}^2 * N_T$

Lastly, a database implementing the granular partitioning technique (d) can materialize only the columns required, since data is stored column-wise. Regarding number of records, there are two possible cases: (a) if the query filter matches exactly a partition boundary, then only records matching the filter will be scanned and $\delta = 1$, where δ represents the amount of data outside the query scope but inside partitions that intersect with the query filter, or (b) if the filter intersects a partition boundary, then records outside the query scope will be scanned and δ is proportional to the number of records that match that specific dimension range. Insertions are done in C_T time by calculating the correct *pid* and appending each column to the back of the record list and space complexity is proportional to $C_T * N_T$ since no other auxiliary data structures are stored.

4.5 Related Work

Indexing and processing highly dynamic datasets at scale has been an active subject of research in the past decade. Key-value stores have been tremendously successful in achieving a high number of updates and queries per second, at the cost of sacrificed

functionality [109] [110] [111] [112]. Most of these data stores offer quick access over a single key and ordered key iteration, but lack support for multi-key access and have limited aggregation capabilities. In addition, a few traditional databases [113] provide some type of multidimensional indexing primitives, but are unable to interactively handle highly dynamic datasets.

Scalable data processing frameworks such as *MapReduce* [114] and more recently Spark [115] have also become very popular, but they mostly focus on batch processing and do not offer any native multidimensional indexing capabilities. Nishimura *et al.* proposed a framework called *MD-HBase* [116] that builds multidimensional indexes on top of MapReduce using K-D Trees and Quad Trees, in such a way that these indexes can be used to quickly evaluate query filters. Due to the use of non-constant update time data structures, the number of records ingested per second as reported by their experiments is 1 to 1.5 orders of magnitude lower if compared to our tests, in equivalent hardware (see Chapter 6). In addition, their work focus on location aware service, which are known to have low dimensionality.

The use of linearization to transform a multidimensional space into one-dimensional has been used recently in a few related papers. Jensen *et al.* [117] has used Z-ordering and Hilbert curves as space filling curves in order to build a one dimension B+Tree index leveraging the linearized one-dimensional values. Patel *et al.* [118] have developed an indexing method named STRIPES that supports efficient updates and queries at the cost of higher space requirements. The focus of these two previous works is to index dynamic positional datasets generated by moving objects. Additional research has been done regarding multidimensional indexes to track moving objects [119] [120] [121], also called *spatio-temporal* datasets, but their are orthogonal to our work due to the limited dimensionality and workload characteristics.

In addition, recent work has been done towards making multidimensional indexes more scalable. Wang *et al.* [122] proposed the creation of a two-level multidimensional index comprising a global and a local index. The entire space is then broken down into several smaller spaces in such a way that the global index keeps tracks of spaces and the local

index efficiently store local data point. In this work, Wang *et al.* proposes the creation of R-Tree based local stores over a Content Addressable Network (CAN) of local stores, while Zhang *et al.* [123] proposes the creation of a hierarchy containing an R-Tree as the top level and K-D Trees for local indexing.

Lastly, Tao *et al.* [124] proposed an efficient indexing technique to answer nearest neighbor queries over high dimensional data. Their approach uses a locality sensitive hashing to reduce the dimensionality and then apply a Z-ordering to linearize the dataset, which further will be queried leveraging a one-dimensional B-Tree index.

In addition to research prototypes and contributions to particular aspects of multi-dimensional indexing, there are also top-down system implementations that support a similar data model or organize data in a similar manner to the approach described in this work. The systems described in the following paragraphs have been successfully commercialized or are well-known open source projects.

SciDB [125] is an array database with a similar multidimensional data model: a user can define arrays composed by a set of dimensions and metrics. Arrays can similarly be chunked into fixed-size strides in every dimension and distributed among cluster nodes using hashing and range partitioning. SciDB, however, focuses on scientific workloads, which can be quite different from regular OLAP use cases. While SciDB offers features interesting for operations commonly found in image processing and linear algebra, such as chunk overlap, complex user defined operations, nested arrays and multi-versioning, Cubrick targets fast but simple operations (like sum, count, max and avg) over very sparse datasets. In addition, SciDB characteristics like non-sparse disk storage of chunks, multiversioning and single node query coordinator make it less suited to our workloads.

Nanocubes [62] is a in-memory data cube engine that provides low query response times over spatiotemporal multidimensional datasets. It supports queries commonly found in spatial databases, such as counting events in a particular spatial region, and produces visual encodings bounded by the number of available pixels. Nanocubes rely on a quadtree-like index structure over spatial information which, other than posing a memory overhead for the index structure, limits (a) the supported datasets, since they need be spatiotem-

poral, (b) the type of queries because one should always filter by spatial location in order not to traverse the entire dataset and (c) visual encoding output.

Despite being a column-store database and hence not having the notion of dimensions, hierarchies and metrics, Google's PowerDrill [126] chunks data in a similar fashion to Cubrick. A few columns can be selected to partition a particular table dynamically, i.e., buckets are split once they become too large. Even though this strategy potentially provides a better data distribution between buckets, since PowerDrill is a column store, data needs to be stored following a certain order, and thus each bucket needs to keep track of which values are stored inside of it for each column, which can pose a considerable memory overhead. In Granular Partitioning, since we leverage fixed range partitioning for each dimension, the range of values a particular *partition* stores can be inferred based on its *partition_id*.

Druid [127] is an open-source analytics data store designed to efficiently support OLAP queries over realtime data. A Druid cluster consists of different types of nodes where each node type is designed to perform a specific set of actions. *Realtime* nodes encapsulate the functionality of ingest and query event streams, being responsible to store data for only a small time range. Periodically, each of these nodes will build an immutable block of data and send to *Historical* nodes for persistent storage. *Broker* nodes propagate the queries to the correct Historical and Realtime nodes and aggregate partial results. Data in persistent storage is organized within *segments* which are partitioned based on a timestamp key. Optionally, Druid supports the creation of bitmap indexes in order to provide fast queries over columns other than the timestamp. Despite providing a similar data model as the one presented on this work, Druid cannot speedup ad-hoc filters against any columns of the dataset.

Apache Kylin [128] is another distributed analytics engine designed to provide a SQL interface and OLAP operations leveraging the Hadoop and HDFS environment. Kylin supports the definition of cubes, dimensions and metrics, but differently from the proposed approach, pre-aggregations are calculated beforehand using map-reduce jobs (due to the high number of possible aggregations) and saved into an HBase cluster in the form of

key-value pairs. At query time, the correct pre-aggregations are retrieved and the query results are computed. However, due to the heavy use of pre-calculations, this architecture is not optimized for highly dynamic datasets.

CHAPTER 5

CUBRICK DESIGN

In this chapter we present the design and architecture of Cubrick, a novel distributed in-memory OLAP DBMS specially suited for low-latency analytic queries over highly dynamic datasets. Cubrick organizes and indexes data using the Granular Partitioning technique described in Chapter 4.

In order to operate at the scale required by the target workloads, rather than implementing the Granular Partitioning technique inside a popular open source DBMS (such as MySQL or PostgreSQL [34] [36]) and leverage all the features that would come with them, we took a different approach; instead, we wrote a brand new architecture from scratch focused on simplicity and only added the features required to support our use cases. That means that several well-known database features such as support for updates and intra-partition deletes, local persistency, triggers, keys and any other database constraints are missing, but only the essential features needed for loading de-normalized multidimensional datasets and execute simple OLAP queries are provided. However, even though we have built the system from the ground up, Cubrick currently is a full-featured DBMS and supports multiple production workloads at Facebook.

As described in Section 4.1, the input for Cubrick are de-normalized multidimensional tables composed of dimensions and metrics, either coming from realtime data streams or batch pipelines. All dimensional columns are part of the partitioning key and stored using a encoding technique called *BESS* encoding (*Bit-Encoded Sparse Structure* [129]). Metrics are stored column-wise per partition and can optionally be compressed.

Once loaded, Cubrick enables the execution of a subset of SQL statements mostly limited to filters, aggregations and groupings. Since Cubrick's main focus is query latency, all data is stored in memory in a cluster of shared-nothing servers and all queries are naturally indexed by organizing the data using Granular Partitioning. Finally, since Cubrick's

primary use case are real time data streams, we make heavy use of multiversioned data structures and copy-on-write techniques in order to guarantee that insertions never block queries.

5.1 Terminology

In this section, we define the terminology we use to describe Cubrick throughout this work. Cubrick *cubes* implement the same abstraction as regular database relations, or an unordered set of tuples sharing the same attribute types. In practice, a cube is equivalent to a database table. Since Cubrick exposes a multidimensional view of the dataset, we refer to each tuple inside a cube as a *cell*, rather than as rows or records. Each attribute inside a particular cell is either a *dimension* or a *metric*, where dimensions are attributes used for filtering and usually have a lower cardinality, and metrics are attributes used in aggregation functions, generally with higher cardinality. Lastly, we denote the set of dimensions of a particular cell as being its *coordinates*.

Considering that Cubrick leverages the Granular Partitioning technique that partitions the cube by ranges in every single dimension, the resulting partitions are, in fact, smaller cubes – or precisely, *hypercubes*. We refer to each of these smaller cubes as *bricks*. Additionally, a partition id is associated to each brick, which we refer to as brick id or *bid*. Since the list of bricks for a particular cube is stored in a sparse manner, we call an *active brick* a partition that contains at least one cell and is allocated in memory.

5.2 Data Organization

Cubrick organizes and stores all data in-memory inside partitions we refer to as bricks. Each brick contains one or more cells — bricks containing zero records are not represented in memory — and is responsible for storing a combination of one range for each dimension. Thus, the number of possible bricks can be calculated by the product of the number of ranges defined for each dimension. In practice, the number of active bricks at any given time is considerably lower since our datasets are not evenly distributed and do not exercise

the full cardinality. Another consequence of this fact is that the number of cells stored by each brick can have a high variance (more about effects of data distribution in Section 5.6).

Within a brick, cells are stored column-wise and in an unordered and sparse fashion. Each brick is composed of one array per metric plus one array to store a bitwise encoded value to describe the in-brick coordinates using a technique called Bit Encoded Sparse Structures, or *BESS* [129]. BESS is a concatenated bit encoded buffer that represents a cell coordinate's offset inside a particular brick on each dimension. At query time, based on the in-brick *BESS* coordinates and on the *bid*, it is possible to reconstruct the complete cell coordinate while keeping a low memory footprint. The number of bits needed to store a cell's coordinate (*BESS*) for a particular cube containing n dimensions where R_k is the range size of the *k-th* dimension is:

$$\sum_{d=1}^{n} \lceil \lg (R_d) \rceil$$

For all dimensions composed of string values, an auxiliary hash table is maintained to associate labels to monotonically increasing ids — a technique also called *dictionary encoding* [130]. Internally, all data is stored based on these ids and only converted back to strings before returning data to users.

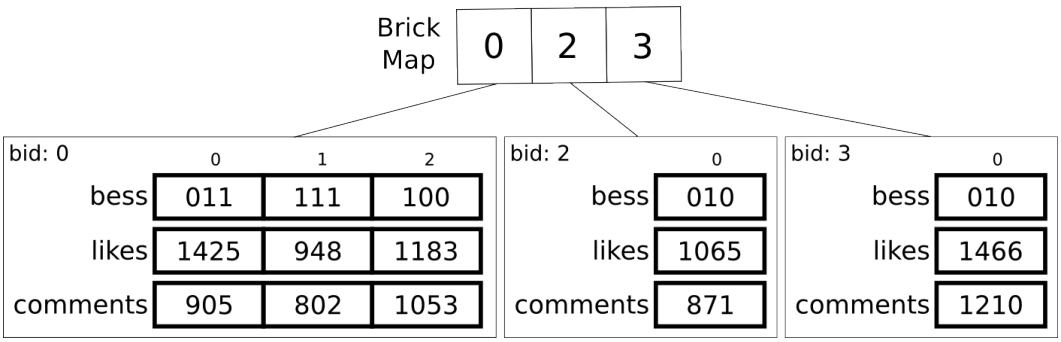

Figure 5.1: Cubrick internal data organization for the dataset presented in Table 4.1.

Figure 5.1 illustrates how Cubrick organizes the dataset shown in Table 4.1. Once

that dataset is loaded, three bricks are created and inserted into *brick map*: 0, 2 and 3, containing 3, 1 and 1 cell, respectively. Each brick is composed by three vectors, one for each metric (*Likes* and *Comments*) and one to store the BESS encoding. The BESS encoding for each cell represents the in-brick coordinates and on this example can be stored using three bits per cell — two for region, since range size is 4, and 1 for gender since range size is 2.

Adding a new cell to a brick is done by appending each metric to the corresponding metric vector and the BESS (calculated based on the new cell's coordinates) to the BESS buffer. Therefore, cells are stored in the same order as they were ingested, and at any given time cells can be materialized by accessing a particular index in all metric vectors and BESS buffer.

Deletes are supported but restricted to predicates that only match entire bricks in order to keep the internal brick structures simple. Cell updates are not supported since it is not a common operation in any of our use cases. In our current usage scenarios, updates are only required when part of the dataset needs to be re-generated. In these cases, the new dataset is loaded into a staging cube so that all bricks can be atomically replaced.

5.3 Distributed Model

In addition to the cube metadata that defines its schema, *i.e.*, set of dimensions and metrics, at creation time a client can also specify the set of nodes that should store data for a particular cube. Alternatively, the client can also only specify the number of nodes. If the number of nodes that will store data for a cube is bigger than one, each node is hashed and placed into a consistent hashing ring [8] and data is distributed among the nodes by hashing each brick id into this ring and thereby assigning bricks to nodes.

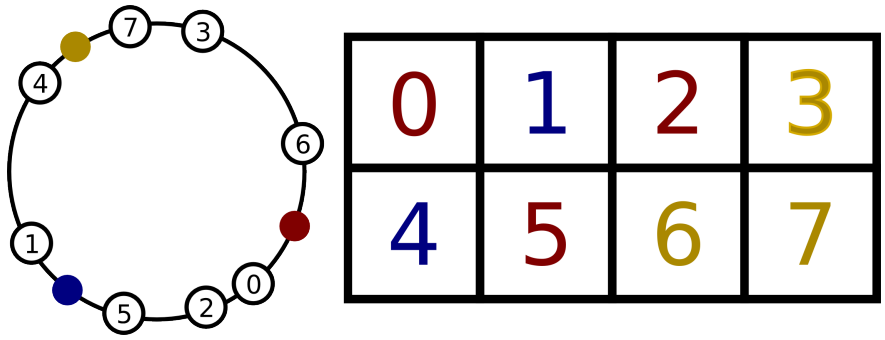

Figure 5.2: Consistent hashing ring that assigns bricks to nodes in a distributed 8-brick cube. The three nodes are represented by the solid circles in the ring.

Figure 5.2 illustrates a cube composed of 8 bricks distributed between 3 nodes. In this example, after all nodes and brick ids have been hashed and positioned in the hash circle, bricks 0, 2 and 5 are assigned to the node in red, 1 and 4 to the node in blue, and 7, 3 and 6 to the node in yellow.

Each node only keeps track of the bricks assigned to it. Therefore, queries need to be broadcasted to all nodes since every node could potentially store a brick that could match the query filters. After receiving the partial results from all nodes involved in the query the results are aggregated and returned back to the client. Ultimately, data replication and failure tolerance is achieved by ingesting the same dataset to multiple Cubrick clusters.

In a previous version of Cubrick we used to support a user supplied *segmentation clause* per cube that would specify how data is distributed throughout the cluster (based on the value of one or a few dimensions). The rationale is that by grouping bricks in the same *segment* of a cube together, a few queries could only be forwarded to a subset of the nodes, given that its filters match the segmentation clause. We ended up dropping support for this since: (a) user defined segmentation clause usually introduces more data skew between nodes than hashing all active bricks, (b) very few queries actually had filters that matched exactly the segmentation clause so most queries were forwarded to all nodes anyway, and (c) worsening of *hot spots* since distribution of queries per segment is usually not even.

5.4 Query Engine

Cubrick supports only a subset of SQL queries, or precisely the ones composed of filters (or select operators, in order to define the search space), aggregations and *group bys* (pivot). Other minor features such as *order by*, *having* clauses and a few arithmetic and logical expression are also supported, but they interfere less in the query engine since they are applied *pre* or *post* query execution.

Nested queries are not supported in any form, and considering that the loaded dataset is already de-normalized, there is also no support for joins. These assumptions make the query engine much simpler since there is only one possible query plan for any supported query. In fact, these assumptions remove the need for a query optimizer whatsoever, considerably simplifying the query engine architecture.

All queries in a Cubrick cluster are highly parallelizable and composed of the following steps:

Propagate query to all nodes. The query is parsed from a SQL string to a intermediary representation based on Thrift [131], which is an RPC and serialization framework heavily used at Facebook, and propagated to all nodes in the cluster that store data for the target cube.

Parallelize local execution. The list of local active bricks is broken down into segments and assigned to multiple tasks (threads of execution), in a way that each task has a subset of the total number of active bricks to scan.

Allocate buffers. An advantage of knowing beforehand the cardinality of each dimension is that the *result set cardinality* is also known. For instance, a query that groups by the dimension *Region* from the example shown in Figure 4.1, is expected to have an output of, at most, 8 rows — the maximum cardinality defined for that dimension. In cases where the result set cardinality is known to be small, a dense vector can be used to hold and merge partial results more efficiently than the widely used hash maps (STL's unordered_maps) due to better memory access patterns. In practice it is usually more efficient to leverage a dense vector for most one dimensional *group by's*, whereas hash maps are used mostly in multidimensional group by's, where the output cartesian cardi-

nality can be large. Based on empirical queries from our use cases, we have observed that the vast majority of queries group by one column or none; therefore, this optimization is crucial to save CPU cycles and reduce query latency. Figure 5.3 shows a comparison of the query execution times using dense and sparse buffers.

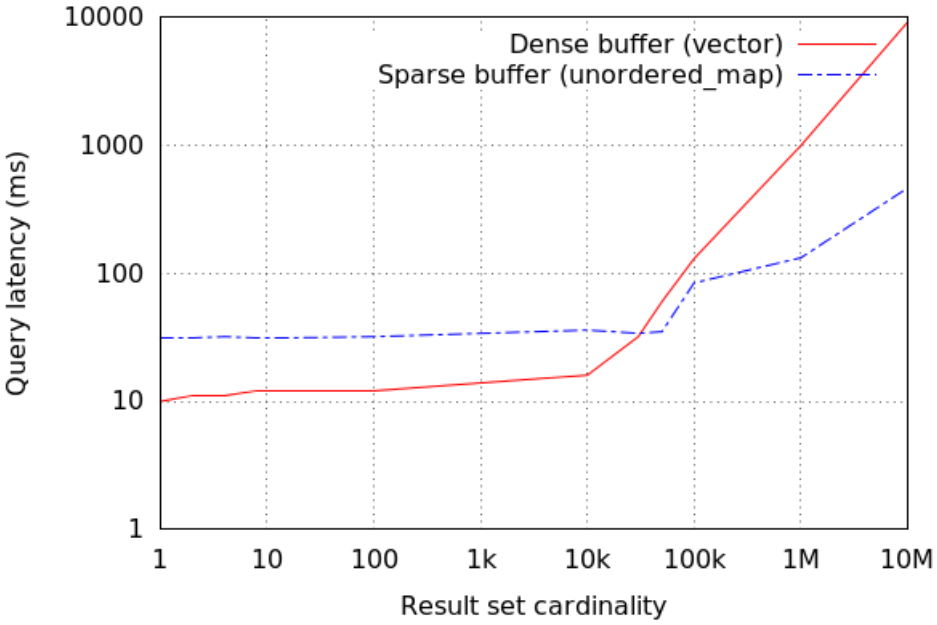

Figure 5.3: Query latency for different result set cardinalities using sparse and dense buffers.

Finally, based on the expected result set cardinality and the operation to be executed, each task decides the more suitable buffer type to allocate. Operations such as *sum*, *count*, *min* and *max* are executed using a single value per possible result set element (whether using dense vectors or hash maps), whilst operations like *pct*, *count_distinct*, and *topK* require one hash map per result set value.

Scan/prune and generate partial results. Once all buffers are allocated, each task matches every brick assigned to it against the query filters and decides whether it needs to be scanned or can be safely skipped. In case the brick needs to be scanned, the operation is executed and merged to the task's buffer; otherwise, the next brick is evaluated.

Aggregate partial results. Partial results from completed tasks and remote nodes

are aggregated in parallel as they are available, until all tasks are finished and the full result set is generated.

Translate labels and format. The ids used internally present in the result set are translated back to labels — this procedure is equivalent to decoding the dictionary encoded values. Since the result sets returned from queries are usually small if compared to the dataset size (unless the group by clause contains several columns), the overhead of looking up the dictionary encoding hash tables is negligible. Lastly, the result set is translate from the internal representation to the appropriate format requested by the user (currently CSV or JSON).

5.5 Rollups

One interesting feature of storing data sparsely and in an unordered fashion inside bricks is that nothing prevents one from inserting multiple distinct cells in the exact same coordinates. Even though conceptually belonging to same point in space, these cells are stored separately and are naturally aggregated at query time in the same manner as cells with different coordinates. In fact, since there is no possible filter that could match only one of the cells and not the other(s), they can be safely combined into a single cells containing the aggregated metrics.

This operation, which we refer to as *rollup*, can be set on a per-cube basis on a Cubrick cluster and consists of a background procedure that periodically checks every brick that recently ingested data and merges cells containing the same coordinates. Considering that the in-brick coordinate for each cell is encoded and stored in the *BESS* buffer, rolling up a brick is a matter of combining cells with the same *BESS* value. Table 5.1 illustrates an example of a two-dimensional dataset containing multiple cells per coordinate, before and after a rollup operation.

Region	Gender	Likes	Comments
Before Rollup			
CA	Male	1	0
CA	Male	0	1
CA	Male	1	1
CA	Female	1	0
CA	Female	1	3
After Rollup			
CA	Male	2	2
CA	Female	2	3

Table 5.1: Sample data set before and after a rollup operation.

A compelling use case for rollup operations is changing the granularity in which data is aggregated at ingestion time without requiring external aggregation. In this way, data for a particular cube can be ingested from a more granular data source — or one containing more dimensions —, and any extra dimensions not defined in the cube can be discarded. Once ingested, the cube will contain as many cells as the base data set but will eventually get compacted by the rollup procedure. Important to notice is that due to the very associative and commutative nature of these aggregations, queries executed prior, during or after the rollup operation will generate identical results.

Finally, this operation is particularly useful for real time stream data sources, where data is generated at the lowest granularity level (*e.g.* user level) and the aggregations are generated partially as the data flows in, transparent to user queries.

5.6 Data Distribution

One of the first questions that arise when dealing with statically partitioned datasets is *what is the impact of data distribution in query performance and overall memory usage?* The intuition is that the best performance is achieved by perfect evenly distributed datasets, where the number of cells stored within each partition is the same, and slowly degrades as the skew between partitions increase. This comes from the fact that scanning too many small partitions can be a burdensome operation due the lack of memory locality.

Unfortunately, most Facebook's datasets are skewed. For instance, a dataset containing the number of posts per demographic is likely to be denser close to countries with

high user population and more sparse towards other countries. The same effect is observed for other demographic dimensions, such as region, age, gender, marital and educational status.

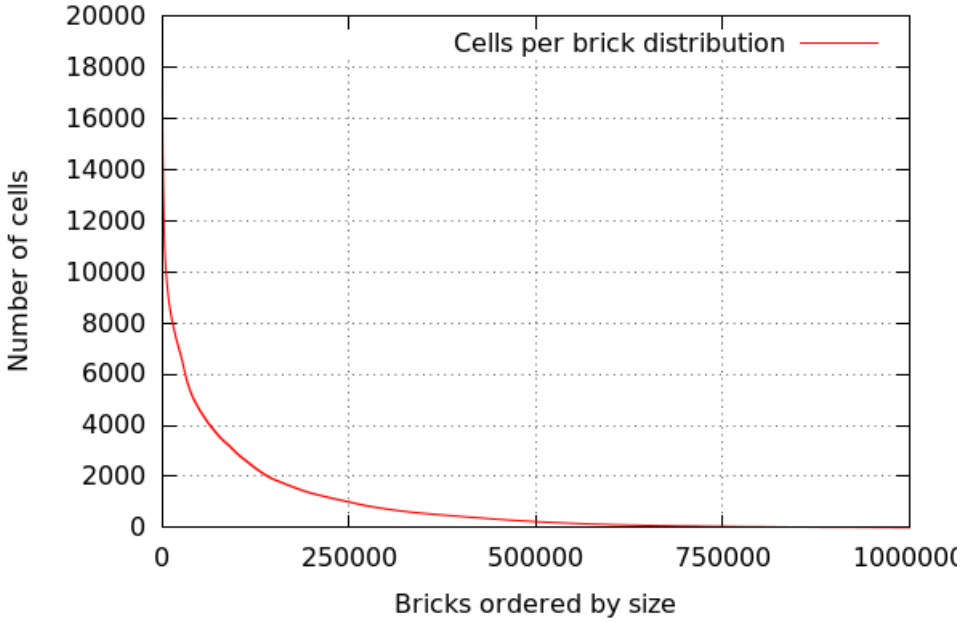

Figure 5.4: Distribution of cells per brick in a real 15-dimensional dataset.

To illustrate the distribution of a real dataset, Figure 5.4 shows the number of cells stored per brick on a 15 dimensional dataset from one of the projects that leverage Cubrick for analytics. For this specific dataset, the distribution of cells between the 1 million active bricks follows a long tail distribution, where a few bricks can contain around 20,000 cells but the vast majority consists of a rather small number of cells.

In order to evaluate the impact of data distribution, we generated two artificial datasets containing one billion records each and loaded into a one node Cubrick cluster. The first dataset is perfectly even, where all bricks contain the exact same number of cells, whilst the second dataset is skewed and follows the same long tail distribution as found in the real dataset illustrated by Figure 5.4. We then loaded each dataset using three different range size configurations in order to generate 10k, 100k and 1M bricks to evaluate how the total number of bricks impacts query performance.

The results of a full scan on these datasets is shown on Table 5.2. For the evenly distributed dataset, as one would expect, query latency increases the more bricks Cubrick needs to scan, even though the number of cells scanned are the same. We attribute this fact to the lack of locality when switching bricks and other initialization operations needed in order to start scanning a brick. However, the query latency increase is relatively small (around 17% from 10k to 1M) if compared to the increased number of bricks — 100 times larger.

We observed an interesting pattern when analyzing the skewed distribution results. A similar effect can be seen for the 100k and 1M bricks test where query latency increases the more bricks are required to scan, but curiously, query latency for the 10k bricks test is significantly larger than for 100k. When digging further into this issue, our hypothesis was that some threads were required to do more work (or scan more cells), and queries are only as fast as the slowest thread. In order to evaluate the hypothesis, we have also included the coefficient of variation of the number of cells scanned per threads in Table 5.2. We observe that the variation is significantly higher (0.75%) for the 10k brick test than all other tests, since the small number of bricks and coarse granularity make it difficult to evenly assign bricks to different threads.

Lastly, regarding the impact of data distribution in memory usage, we have seen that the memory footprint difference from 10k to 1M bricks is below 1%, under both even and skewed distributions.

# of bricks	Even	Skewed
10k	456ms / 0.039%	605ms / 0.73%
100k	478ms / 0.003%	482ms / 0.06%
1M	536ms / 0.001%	572ms / 0.01%

Table 5.2: Full table scan times for a 1 billion cells dataset following a perfectly even and skewed distribution (as shown in Figure 5.4) using different range sizes to obtain different numbers of bricks. Also showing the coefficient of variation (stddev / median) of the number of cells scanned per thread.

CHAPTER 6

EXPERIMENTAL RESULTS

In this chapter we present an experimental evaluation of our current implementation of Cubrick. Cubrick has been written from the first line of code in C++11, leveraging an RPC framework extensively used at Facebook called Thrift [131]. Thrift is used to define all remote methods and data types available, in addition to serialization functions and thread management primitives. Thrift is used for both the communication between client and server as inter-server communication when running in cluster mode.

All servers used in the test environment contain 32 logical CPUs and 256 GB of memory, being interconnected by regular 10Gb Ethernet networks. Even though we have used different amounts of servers for different experiments, the servers were always located in the same datacenter and positioned close together in such a way to keep low latency. Round-trip latency between any two servers is usually about 0.1 ms under normal network conditions, as measured by a regular ping command.

All experiments were executed directly on the physical servers (without using any kind of virtualization) running a linux distribution based on CentOS. All servers were also exclusively allocated for these experiments in order to reduce external interference to the results. Since the environment was fairly isolated, test results were consistent and overall variation was below 1% on all experiments. Due to the very low variation on experimental results, all values presented are the average of only three executions unless noted otherwise.

The dataset used in these experiments belong to a Facebook project that leverages Cubrick for analytics. This dataset is composed by 26 columns, from which 16 are dimensions and 10 are metrics. From the 16 dimension columns, 10 of them have low cardinality (less than 16 distinct values), and the other 6 can have up to thousands of distinct values. All metrics are stored as 32 bits unsigned integer data types.

The dataset size presented on all experiments is related to the in-memory size of the data after being loaded into Cubrick. This size usually does not reflect this datasets' size on other data storage systems that might implement different data encoding and compression techniques. For instance, this specific dataset takes about 50 bytes of memory per row on a Cubrick cluster, but about 120 bytes per row if exported as a CSV file.

The experiments are organized as follows. We first compare how efficiently Cubrick and two other DBMS architectures — namely, a row-oriented and a column-oriented DBMS — can prune data out of a query scope when filters with different restrictivity are applied. Further, we show absolute latency times for queries containing different filters over a dataset spread over different cluster sizes. Finally, we present the results of a series of experiments regarding data ingestion, showing absolute ingestion rates against CPU utilization, the impact of ingestion in query latency and lastly the resources required by the rollup operation.

6.1 Pruning Comparison

The purpose of this experiment is to measure how efficiently Cubrick can prune data from queries without relying on any other auxiliary data structures, in comparison to other DBMS architectures. In order to run this test, we loaded a dataset containing 1 billion rows into a single node Cubrick cluster, resulting in about 100 GB of memory utilization. Further, we collected several OLAP queries from real use cases, containing exact match and range filters over several dimension columns. We intentionally collected queries whose filters match different percentages of the dataset in order to test the impact of data pruning in query latency. We have also executed the same queries against two well known commercial DBMSs; a row-store (MySQL) and a column-store (HP Vertica).

Since Cubrick is a in-memory DBMS optimized for interactive latency, measuring absolute query latency against other disk based systems is not a fair comparison. Rather, in order to only evaluate how efficiently different architectures prune data at query time we compared the latency of each query with the latency of a full table scan (query touching the entire table without any filters) and show the ratio between them.

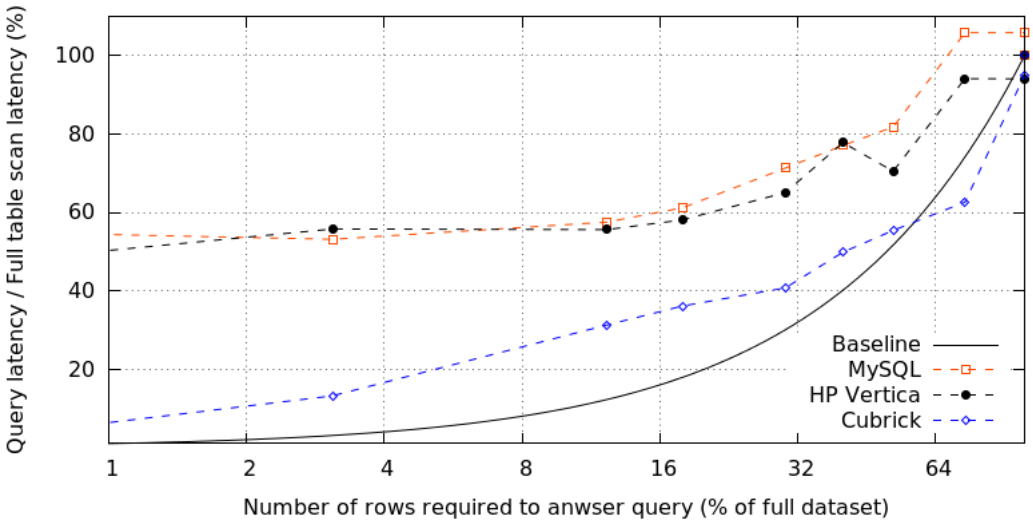

Figure 6.1: Relative latency of filtered OLAP queries compared to a full table scan.

Figure 6.1 shows the results. The x-axis shows the percentage of the dataset the queries touch (log scale) and the y-axis show the query latency compared with a full table scan. We have also included a baseline that illustrates a hypothetical DBS able to *perfectly* prune data, *i.e.*, where the query latency is exactly proportional to the amount of data required to scan to answer a query. The left side of the chart represents queries that only touch a few records, whilst the extreme right side contains latency for queries that scan almost the entire dataset, and thus are closer to 100%.

For all test queries executed, Cubrick showed better prune performance than the other architectures and performed closer to the ideal baseline hypothetical DBS. The biggest differences were observed when the queries required a very small percentage of the dataset. In both MySQL and Vertica in order to execute a query touching about 3% of the dataset, the time required was slightly above half of the time required to execute a full table scan. On Cubrick, the time required for this query was about 14% of a full table scan. Curiously, Cubrick even performed better than the hypothetical DBS in one of the queries. We attribute this to the fact that the filter applied on this particular query matched exactly brick boundaries, in such a way that the filter did not need to be pushed down to the cell level, thus saving CPU cycles and reducing query latency.

6.2 Queries

In this Section, we evaluate the performance of queries over different dataset sizes and cluster configurations. We focus on showing how efficiently filters are applied and their impact on latency, as well as how Cubrick scales horizontally as more nodes are added to the cluster.

6.2.1 Dataset Size

For the first experiment, we present the results of queries on a dataset containing around 30 columns under four different configurations controlled by how much historical data we have loaded for each test. We show results for 10GB, 100GB, 1TB and 10TB of overall size on a 72 node cluster. In order to achieve low latency for queries, data was not compressed; however, due to Cubrick's native *BESS* and dictionary encoding for strings, each record required only about 50 bytes such that the 10TB test comprised about 200 billion records. For each configuration, we show the latency for a full table scan (non-filtered query), followed by three queries containing filters that match 50%, 20% and 1% of the dataset. These percentages are calculated based on a *count(*)* using the same filters, compared to the total number of loaded records.

Figure 6.2 presents the results. For the first two datasets, 10GB and 100GB, the amount of data stored per node is small so query latency is dominated by network synchronization and merging buffers. Hence, full scans are executed in 18ms and 25ms, respectively, even though the second dataset is 10 times larger. For the following two datasets, the amount of data stored per node is significantly larger, so the latency difference for the filtered queries is more noticeable. Ultimately, in a 72 node cluster, Cubrick was able to execute a full scan over 10TB of data in 763ms, and a query containing filters that match only 1% of the dataset in 38ms.

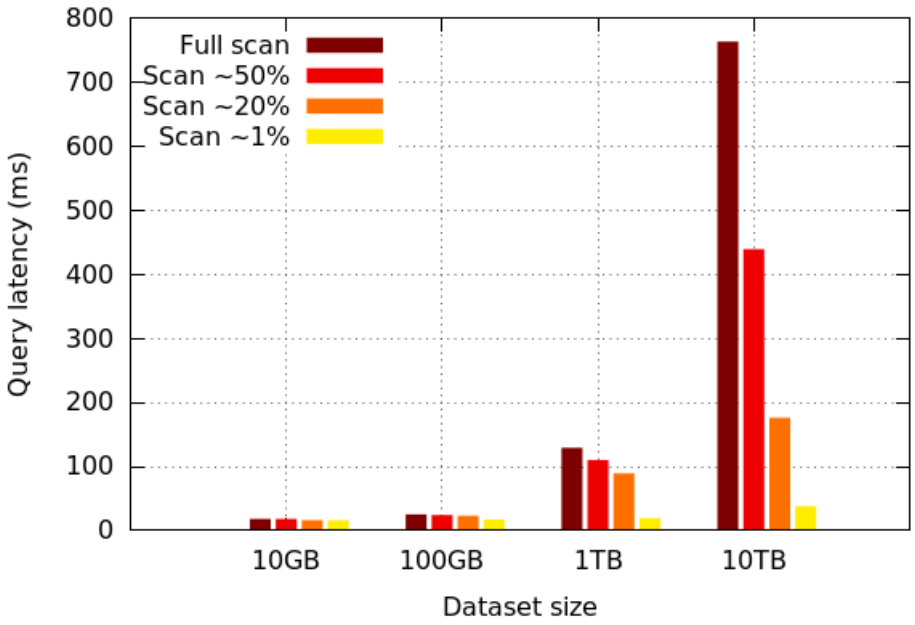

Figure 6.2: Queries over different dataset sizes on a 72 node cluster. The first query is non-filtered (full scan) and the following ones contain filters that match only 50%, 20% and 1% of the dataset.

6.2.2 Horizontal Scaling

For the second test, we focus on horizontal scaling by loading the same dataset in different cluster configurations. We have used two datasets, one sized 150GB (in order to fit in one server's memory) and one sized 1TB. The two datasets were loaded in clusters of four sizes: a single node cluster, a 10 node, a 50 node and a 100 node cluster. Three queries were executed against each configuration: (a) a full table scan, (b) a query touching only 30% of the dataset and (c) a query touching about 1% of the dataset. Note that there are no results for the 1TB dataset in the single node cluster since it does not fit in memory.

Figure 6.3 presents the results. For the 150GB dataset it is possible to observe that query latency highly decreases when the cluster size grows from 1 to 10 nodes. For the other two cluster configurations (50 and 100) the gains are less noticeable since query latency starts being dominated by network synchronization and merging too many remote buffers. The 1TB dataset, on the other hand, still benefits from enlarging the cluster from 50 to 100 nodes since the amount of data that needs to be scanned is larger. In summary, in this experiment we were able to see a 24x speedup in query latency for the 150GB dataset when comparing a single node cluster to the 100 nodes cluster (1296ms and 53ms,

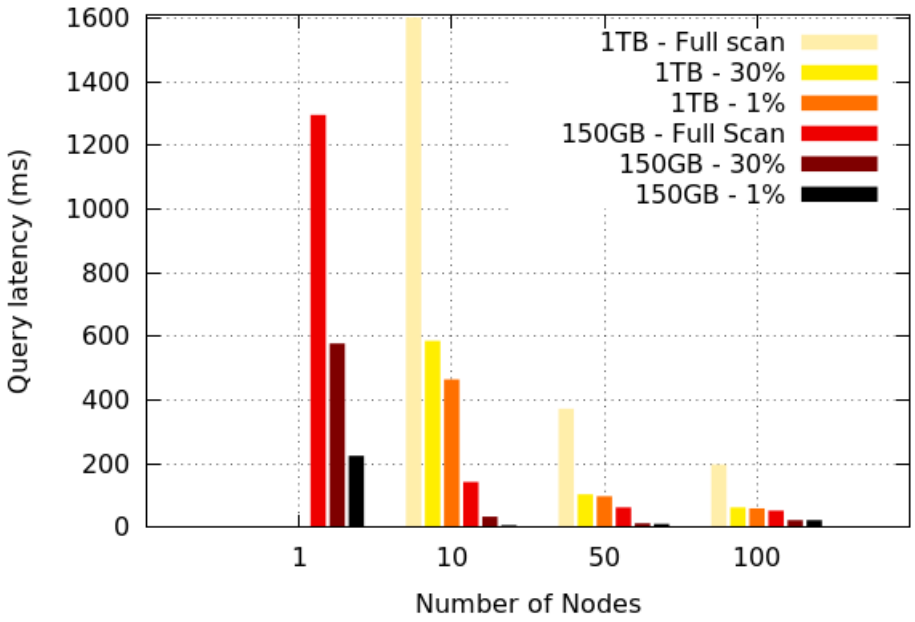

Figure 6.3: Three queries against two datasets under four different cluster configurations.

respectively). For the 1TB dataset, the speedup observed was 8x when comparing 10 to 100 nodes cluster (1602ms and 198ms, respectively).

6.2.3 Filtering

In the third experiment, we have focused on evaluating how efficiently filters over different dimension are applied. We have leveraged the same dataset as the previous experiment — 10TB over 72 nodes — and executed queries containing filters over different dimensions. We randomly took 10 dimensions from this dataset and selected valid values to filter on, in such a way that each different filter required to scan a different percentage of the dataset.

Figure 6.4 shows the results. We have named each column from *dim1* to *dim10* (x-axis), and show the latency of each query as a dot in the column the query had a filter on. We have also measured the percentage of the dataset each filter required the query to scan and show in it a color scale, being a dark dot close to a full table scan and a yellow a query that only scans a small part of the dataset. The top black line at 763ms is the hypothetical upper bound, since it is the time required to execute a full table scan over the 10TB dataset.

From these experimental results, we observe a pattern that goes from yellow to black

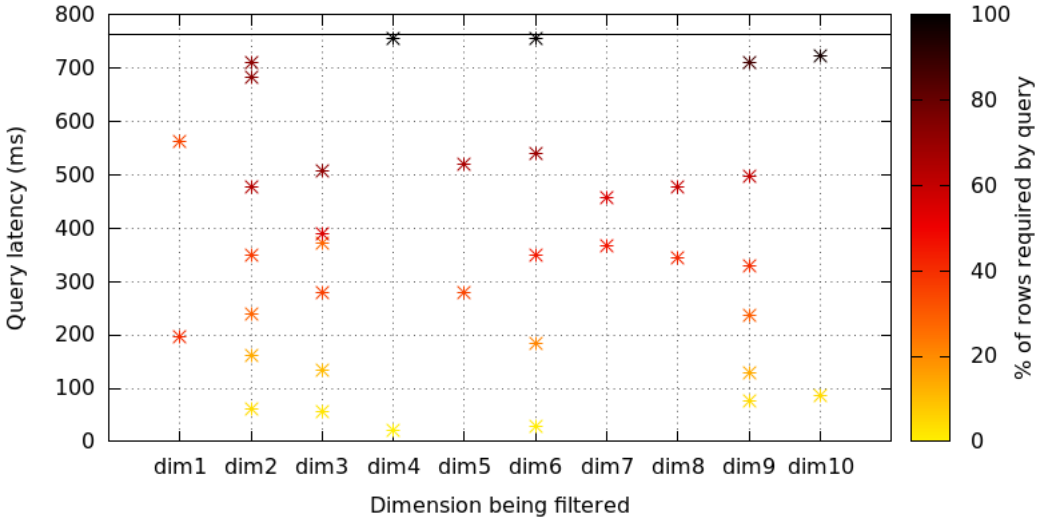

Figure 6.4: Latency of queries being filtered by different dimensions over a 10TB dataset on a 72 node cluster.

as the query latency increases, independent of the dimension being filtered — the value for x. This fact supports our hypothesis that filters can be efficiently applied on every dimension of the dataset, in such a way that query latency is driven by the amount of data that needs to be scanned, rather than the column on which to filter.

6.3 Ingestion

In this section, we evaluate how fast Cubrick can ingest data as well as what is the impact of data ingestion on overall cluster resource utilization and ultimately in query latency. The rationale behind the organization of these experiments relies on the fact that Cubrick is built on top of lock-free data structures and copy on write containers, in such a way that insertions never block queries. Therefore, queries are reduced to a CPU problem since there is no lock contention for queries and all data is already stored in-memory. Also, network synchronization required is minimal and only used to collect partial results from other nodes.

We start by measuring the resource consumption for ingestion and the rollup procedure (which is used in most real time use cases), in order to evaluate how much CPU is left for query processing, and conclude by presenting the impact of ingestion on actual query execution and query latency.

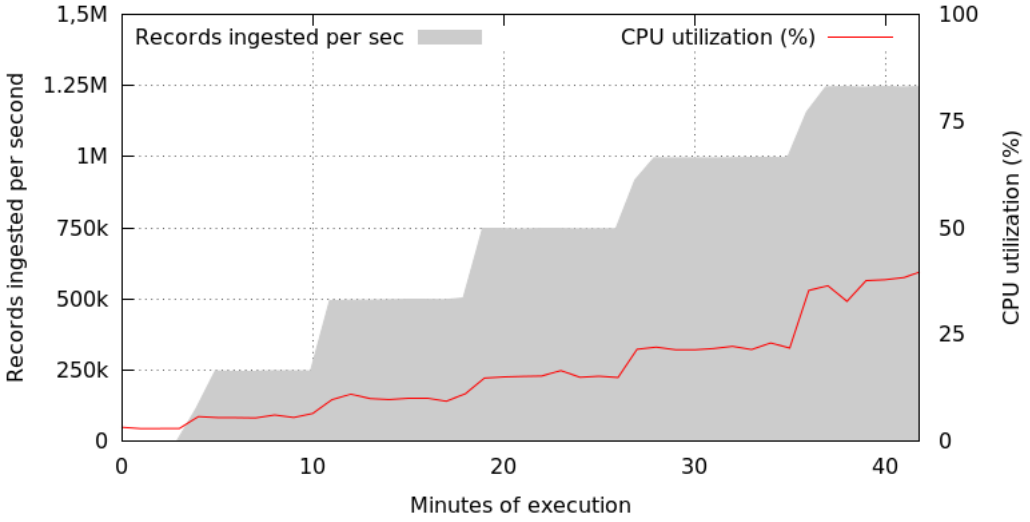

Figure 6.5: Single node cluster CPU utilization when ingesting streams with different volumes.

6.3.1 CPU Overhead

In order to evaluate the CPU overhead of ingestion in Cubrick, we created a single node cluster ingesting a stream containing real data, and slowly increased the volume of that stream in order to measure the overall impact on CPU utilization. The stream volume started at 250k records per second and increased at 250k steps after a few minutes.

Figure 6.5 presents the results. From this experiment, we can see that CPU utilization is minimal up to 500k records per second (about 15%), but even as the volume increases and approaches 1M records per second, the CPU usage is still relatively low at about 20%. In summary, on a Cubrick cluster ingesting around 1M rows per second *per node*, there is still around 80% of CPU left for query execution.

6.3.2 Rollups

As described in Section 5.5, the rollup operation is extensively used in Cubrick deployments in order to summarize and aggregate datasets being transfered in different aggregation levels. The most common example is a Cubrick cluster storing data aggregated at demographic level, but ingesting data generated at user level. Data is ingested as soon as it arrives and is immediately available for queries. Later, a background procedure aggregates all cells containing the same *BESS* encoding in order to keep low memory footprint.

Figure 6.6: Memory utilization of servers ingesting data in real time with rollup enabled.

Given that Cubrick is a memory-only DBMS and the large volume of ingested streams, the rollup procedure is crucial to keep the dataset within a manageable size.

Figure 6.6 illustrates the memory utilization of a few production nodes ingesting a real stream with rollups enabled. The pattern is similar to a *sawtooth* function, where memory is freed every time the rollup procedure runs and summarizes a few cells, but since new records are continuously being ingested it quickly spikes again while the rollup thread sleeps. Over time, however, the memory utilization tends to slowly grow even after running a rollup since new cells stored in a space of the cardinality not exercised before cannot be rolled up.

6.3.3 Impact on Queries

In this experiment, we measure the impact of ingesting streams of different volumes against query latency. The volume of the input stream used started at 500k records per second and slowly increase to 1M, 2M and close to 3M, which is about the size of the target streams for some internal use cases. We decided to first load a 1 terabyte chunk of the dataset so the dataset size variation due to ingestion during the test is minimal. The data was also being rolledup. We then defined three different queries that were executed continuously during the ingestion test: (1) a full scan that aggregates all metrics with no

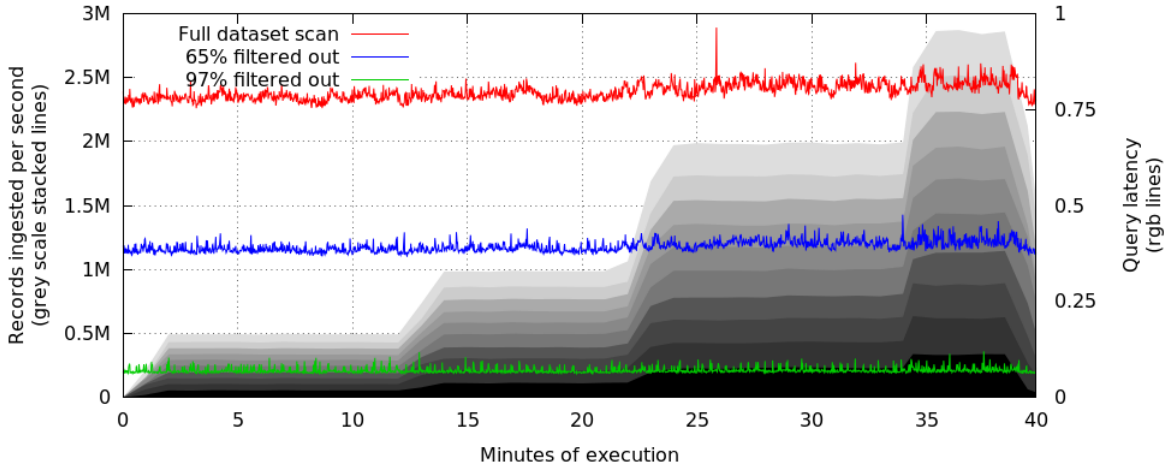

Figure 6.7: Latency of queries on a 10 node cluster ingesting streams of different volumes.

filters, (2) a filtered query that needs to scan about 35% of the dataset and (3) a more restrictive filtered query that requires a scan on only 3% of the dataset.

Figure 6.7 presents the results on a 10 node Cubrick cluster. The filled areas in grey scale on the background represent the amount of records ingested per second per node, while the colored lines shows the latency of each defined query. We observe that as the input stream approaches 3M records per second, the impact on query latency is barely noticeable and below 5%. This result aligns with the experiment shown in Section 6.3.1, considering that in this test each node is ingesting around 300k records per second. In absolute numbers, queries on the 10 node Cubrick cluster ingesting 2 to 3M rows per second run in about 800ms in the worst case (full scan) and achieve better latency the more filters are applied.

CHAPTER 7

CONCLUSIONS

This work presented a novel multidimensional data organization and indexing strategy for in-memory DBMSs called Granular Partitioning. Granular Partitioning was developed to address a new workload we have found at Facebook, characterized by (a) high dynamicity and dimensionality, (b) scale, (c) interactivity and simplicity of queries and (d) presence of ad-hoc query filters, which we found unsuited for most current OLAP DBMSs and multidimensional indexing techniques.

Granular Partitioning extends the traditional view of database partitioning by range partitioning every dimension of the dataset and organizing the data within small containers in an unordered and sparse fashion. Since the cardinality of each dimension is estimated beforehand, each container can be numbered using a row-major function in such a way that the target partition for a record can be trivially calculated as well as any pruning decisions can be made solely based on the partition number. Moreover, this strategy provides indexed access through every dimension for both exact and range filters in addition to low overhead insertions, due to lack of auxiliary data structures maintenance costs.

This work also presented Cubrick, a distributed multidimensional in-memory DBMS for interactive analytics over large and dynamic datasets. More than a research prototype, Cubrick is a full-featured DBMS we have written from scratch for this work, and currently supports multiple production workloads at Facebook. Cubrick leverages the Granular Partitioning indexing technique in order to support a data model composed of cubes, dimensions and metrics as well as efficient OLAP operations such as slice and dice, roll-up and drill-down. Partitions from a Cubrick cube (or *brick*) are assigned to cluster nodes using consistent hashing and data is hierarchically aggregated in parallel at query time.

In order to evaluate the proposed solution, we presented results from a thorough

experimental evaluation that leveraged datasets and queries collected from a few pilot Cubrick deployments. We showed that by properly organizing the dataset according to Granular Partitioning and focusing the design on simplicity, we were able to achieve the target scale of our workload, storing tens of terabytes of in-memory data, continuously ingesting millions of records per second from realtime data streams and still executing sub-second queries.

Future work is geared towards two directions. First, since Granular Partition statically partitions the dataset, some data distribution characteristics can be problematic and result in a large number of small partitions or a small number of large partitions in some specific scenarios. According to our experiments these situations did not surface as a problem in the evaluated datasets, but this is an important issue to tackle as we make Cubrick more generic and expand to new use cases. One of the ideas to investigate is whether the partitioning can be done incrementally as data is ingested, and how the partition numbering function can be adapted to this case.

Second, strategies to ease the extension of dimensional cardinality need to be investigated. We have described the trivial technique of re-numbering all partitions when a dimension in re-sized, but that can be proven costly if the dataset is also saved on disk for disaster recovery. Ultimately, another avenue for future work is how to extend the numbering function in order to support creation of new dimensions after the cube has been defined.

BIBLIOGRAPHY

[1] Jakob Nielsen. *Usability Engineering.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[2] R. Kimball. *, The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses.* John Wiley & Sons, 1996.

[3] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? *SIGMOD*, Vancouver, Canada, 2008.

[4] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel R. Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alexander Rasin, Nga Tran, and Stan B. Zdonik. C-store: A column-oriented dbms. *VLDB*, pages 553–564, Trondheim, Norway, 2005.

[5] MicroStrategy Inc. Microstrategy olap services. `https://www2.microstrategy.com/software/products/OLAP-services/`, 2016.

[6] Oracle Inc. Oracle essbase. `http://www.oracle.com/technetwork/middleware/essbase/overview`, 2016.

[7] Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, and Yannis Kotidis. Dwarf: Shrinking the petacube. SIGMOD'02, pages 464–475, New York, NY, USA, 2002. ACM.

[8] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[9] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[10] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. *Proceedings of the 31st International Conference on Very Large Databases*, pages 553–564, Trondheim, Norway, 2005.

[11] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). *Proceedings of the 33rd International Conference on Very Large Databases*, pages 1150–1160, University of Vienna, Austria, 2007.

[12] HP Vertica. Vertica - MPP Data Warehouse. `http://www.vertica.com/`, 2016.

[13] EXASolution. EXASOL: The world's fastest in-memory analytic database. `http://www.exasol.com/en/`, 2016.

[14] Ramez A. Elmasri and Shankrant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[15] A.H. Robinson and E. Cherry. Results of a prototype television bandwidth compression scheme. *PIEEE*, 55(3):356–364, March de 1967.

[16] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[17] Sihem Amer-Yahia and Theodore Johnson. Optimizing queries on compressed bitmaps. *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 329–338, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[18] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transaction on Database Systems*, 31:1–38, 2006.

[19] Theodore Johnson. Performance measurements of compressed bitmap indices. *VLDB'99, Proceedings of 25th International Conference on Very Large Databases*, pages 278–289, Edinburgh, Scotland, UK, 1999. Morgan Kaufmann.

[20] David J. Dewitt, Samuel R. Madden, Daniel J. Abadi, Daniel J. Abadi, Daniel S. Myers, and Daniel S. Myers. Materialization strategies in a column-oriented dbms. *In Proceedings of ICDE*, 2007.

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

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

[23] Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD'09, pages 1–2, New York, NY, USA, 2009. ACM.

[24] Erietta Liarou, Stratos Idreos, Stefan Manegold, and Martin Kersten. Monetdb/datacell: Online analytics in a streaming column-store. *Proc. VLDB Endow.*, 5(12):1910–1913, 2012.

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

[26] Marcin Żukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. Tese de Doutorado, Universiteit van Amsterdam, sep de 2009.

[27] Daniel J. Abadi. Query execution in column-oriented database systems. MIT PhD Dissertation, 2008. PhD Thesis.

[28] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[29] Rudolf Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *SIGFIDET Workshop*, pages 245–262, New York, NY, USA, 2002. ACM.

[30] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[31] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 38–49, New York, NY, USA, 1997. ACM.

[32] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communnications of the ACM*, 33(6):668–676, 1990.

[33] MemSQL. MemSQL: The fastest in memory database. `http://www.memsql.com`, 2016.

[34] MySQL. MySQL 5.7 Reference Manual. `https://dev.mysql.com/doc/refman/5.7/en/index.html`, 2016.

[35] Chee-Yong Chan and Yannis E. Ioannidis. Bitmap index design and evaluation. *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 355–366, New York, NY, USA, 1998. ACM.

[36] PostgreSQL. PostgreSQL 9.4.1 Documentation. `http://www.postgresql.org/docs/9.4/`, 2016.

[37] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. Automatic sql tuning in oracle 10g. *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 1098–1109. VLDB Endowment, 2004.

[38] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 1087–1097. VLDB Endowment, 2004.

[39] Surajit Chaudhuri and Vivek Narasayya. An efficient, cost-driven index selection tool for microsoft sql server. *VLDB'97*. Very Large Data Bases Endowment Inc., August de 1997.

[40] Surajit Chaudhuri and Vivek Narasayya. Autoadmin “what-if” index analysis utility. *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 367–378, New York, NY, USA, 1998. ACM.

[41] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. COLT: Continuous On-line Tuning. *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 793–795, New York, NY, USA, 2006. ACM.

[42] Stratos Idreos, Stefan Manegold, and Goetz Graefe. Adaptive indexing in modern database kernels. *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 566–569, New York, NY, USA, 2012. ACM.

[43] Goetz Graefe and Harumi Kuno. Self-selecting, self-tuning, incrementally optimized indexes. *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 371–381, New York, NY, USA, 2010. ACM.

[44] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research*, pages 68–78, Asilomar, CA, USA, 2007.

[45] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The uncracked pieces in database cracking. *Proc. VLDB Endow.*, 7(2):97–108, 2013.

[46] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 5(6):502–513, 2012.

[47] Eleni Petraki, Stratos Idreos, and Stefan Manegold. Holistic indexing in main-memory column-stores. *SIGMOD'15*, Melbourne, Victoria, Australia, 2015. ACM.

[48] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):585–597, 2011.

[49] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–424, Beijing, China, 2007.

[50] Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. Database cracking: fancy scan, not poor man's sort! *Tenth International Workshop on Data Management on New Hardware, DaMoN 2014*, pages 4:1–4:8, Snowbird, UT, USA, 2014.

[51] C. A. R. Hoare. Algorithm 64: Quicksort. *Communication of the ACM*, 4(7):321–, julho de 1961.

[52] MySQL. MySQL 5.7 Reference Manual - Partitioning. `http://dev.mysql.com/doc/refman/5.5/en/partitioning.html`, 2016.

[53] Oracle Inc. Oracle 11g Partitioning Concepts. `http://docs.oracle.com/cd/B28359_01/server.111/b32024/partition.htm`, 2016.

[54] HP Vertica. HP Vertica 7.1.x Documentation. `https://my.vertica.com/docs/7.1.x/HTML/index.htm`, 2016.

[55] J. Orenstein. A dynamic file for random and sequential accessing. *Proceedings of the 9th International Conference on Very Large Databases*, pages 132–141, 1983.

[56] R. H. Guting. Gral: An Extendible Relational Database System for Geometric Applications. *VLDB 89: 15th International Conference on Very Large Databases*, pages 33–44, 1989.

[57] Volker Gaede and Oliver Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[58] Oracle Inc. Oracle Database 12c. `http://www.oracle.com/technetwork/database/database-technologies/`, 2016.

[59] A. Guttman. R-trees: A dynamic index structure for spatial searching. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–54, 1984.

[60] H. Samet. The quadtree and related hierarchical data structure. *ACM Computing Surveys*, pages 187–206. ACM, 1984.

[61] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communication of the ACM*, 18(9):509–517, 1975.

[62] L. Lins, J. T. Klosowski, and C. Scheidegger. Nanocubes for Real-Time Exploration of Spatiotemporal Datasets. *Visualization and Computer Graphics, IEEE Transactions on*, 19(12):2456–2465, 2013.

[63] SQLite. The SQLite R*Tree Module. `https://www.sqlite.org/rtree.html`, 2016.

[64] David Wagner. A generalized birthday problem. *Proceedings of the 22Nd Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO'02, pages 288–303, London, UK, UK, 2002. Springer-Verlag.

[65] Aaron M. Tenenbaum, Yedidyah Langsam, and Moshe J. Augenstein. *Data Structures Using C*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[66] W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. *Proceedings of the Sixth International Conference on Very Large Databases*, pages 212–223, Montreal, Quebec, Canada, 1980. IEEE Computer Society.

[67] Paul Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, 1988.

[68] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing—a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.

[69] Rene De La Briandais. File searching using variable length keys. *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), pages 295–298, New York, NY, USA, 1959. ACM.

[70] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Proceedings of the Eighth International Conference on World Wide Web*, pages 1203–1213, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[71] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[72] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, 2004.

[73] Chee-Yong Chan and Yannis E. Ioannidis. An efficient bitmap encoding scheme for selection queries. *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 215–226, New York, NY, USA, 1999. ACM.

[74] Nick Koudas. Space efficient bitmap indexing. *Proceedings of the Ninth International Conference on Information and Knowledge Management*, CIKM'00, pages 194–201, New York, NY, USA, 2000. ACM.

[75] Gennady Antoshenkov. Byte aligned data compression. Patent US 5363098 A, 1994.

[76] Ekow Otoo Kesheng Wu, Arie Shoshani. Word aligned bitmap compression method, data structure, and apparatus. Patent US 6831575 B2, 2003.

[77] Sebastiaan J. van Schaik and Oege de Moor. A memory efficient reachability data structure through bit vector compression. *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD'11, pages 913–924, New York, NY, USA, 2011. ACM.

[78] Francois Deliege and Torben Bach Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. *In Proc. of EDBT*, pages 228–239, 2010.

[79] M. Vlachos F. Fusco, M. Stoecklin. Net-fli: on-the-fly compression, archiving and indexing of streaming network traffic. *Proceeding of the VLDB'10*, pages 1382–1393, 2010.

[80] Alessandro Colantonio and Roberto Di Pietro. Concise: Compressed 'n' composable integer set. *CoRR*, 1004.0403, 2010.

[81] Daniel Lemire, Owen Kaser, and Kamel Aouiche. Sorting improves word-aligned bitmap indexes. *Data Knowl. Eng.*, 69(1):3–28, 2010.

[82] E. J. Otoo and T. H. Merrett. A storage scheme for extendible arrays. *Computing*, pages 1–9, 1983.

[83] D. Rotem and J. L Zhao. Extendible arrays for statistical databases and olap applications. *Proceedings of SSDBM*, pages 108–117, 1996.

[84] K. Hasan, K. Islam, M. Islam, and T. Tsuji. An extendible data structure for handling large multidimensional data sets. *International Conference on Computer and Information Technology (ICCIT)*, pages 669–674, 2009.

[85] Y. Zhao, P. M. Deshpande, and J. F Naughton. An array based algorithm for simultaneous multidimensional aggregates. *Proceedings of ACM SIGMOD*, pages 159–170, 1997.

[86] J. Nievergelt, H. Hinterberger, and K. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *In Proceedings of the Third ECI Conference*, pages 236–251. Springer-Verlag, 1981.

[87] I. Wald and V. Havran. On building fast kd-trees for ray tracing and on doing that in o(nlogn). *IEEE Symposium on Interactive Ray Tracing*, pages 61–69. IEEE Press, 2006.

[88] J. L. Bentley and Friedman. Data structures for range searching. *ACM Computing Surveys*, pages 397–409. ACM, 1979.

[89] D. B. Lomet and Salzberg. The hb-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, pages 625–658. Morgan-Kaufmann, 1994.

[90] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Computer Graph*, 1980.

[91] H. Fuchs, G. Abran, and E. Grant. Near real-time shaded display of rigid objects. *Computer Graph*, pages 65–72, 1983.

[92] H. Samet and R. Webber. Storing a collection of polygons using quadtrees. *ACM Trans. Graph.*, pages 182–222. ACM, 1985.

[93] J. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[94] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. *International Conference on Very Large Databases (VLDB)*, pages 507–518. VLDB Endowment, 1987.

[95] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD international conference on Management of data*, pages 322–331. ACM, 1990.

[96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The x-tree : An index structure for high-dimensional data. *VLDB'96, International Conference on Very Large Data Bases*, pages 28–39, 1996.

[97] Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, outubro de 1968.

[98] Bradford G. Nickerson and Qingxiu Shi. On k-d range search with patricia tries. *SIAM J. Comput.*, 37:1373–1386, 2008.

[99] Tilmann Zaschke, Christoph Zimmerli, and Moira C. Norrie. The PH-Tree: A space-efficient storage structure and multi-dimensional index. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 397–408, New York, NY, USA, 2014. ACM.

[100] Jair M. Babad. A record and file partitioning model. *Communications of the ACM*, 20(1):22–31, 1977.

[101] Daniel Abadi and Michael Stonebraker. C-store: Looking back and looking forward. Talk at VLDB'15 - International Conference on Very Large Databases, 2015.

[102] Facebook Inc. Scribe: A Server for Aggregating log data Streamed in Real Time. `https://github.com/facebookarchive/scribe`, 2016.

[103] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. *Proceedings of the 12th International Conference*

on Very Large Data Bases, VLDB '86, pages 294–303, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[104] Morton G. M. A computer oriented geodetic database and a new technique in file sequencing. Technical Report Ottawa, Ontario, Canada, 1966.

[105] Giuseppe Peano. Sur une courbe, qui remplit toute une aire plane (on a curve which completely fills a planar region). *Mathematische Annalen*, pages 157–160. Mathematische Annalen, 1890.

[106] Jonathan K. Lawder and Peter J. H. King. Using space-filling curves for multi-dimensional indexing. *Proceedings of the 17th British National Conferenc on Databases: Advances in Databases*, BNCOD 17, pages 20–35, London, UK, UK, 2000. Springer-Verlag.

[107] D Hilbert. Ueber die stetige abbildung einer line auf ein flachenstuck. *Mathematische Annalen*, pages 459–460. Mathematische Annalen, 1891.

[108] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[109] Facebook Inc. Rocksdb: A persistent key-value store for fast storage environments. `http://rocksdb.org/`, 2016.

[110] The Apache Software Foundation. Apache HBase. `https://hbase.apache.org`, 2016.

[111] MongoDB. MongoDB for Giant Ideas. `https://www.mongodb.com`, 2016.

[112] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI'06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[113] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *International Conference on Very Large Data Bases (VLDB)*, pages 1790–1801. VLDB Endowment, 2012.

[114] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[115] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[116] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. *Proceedings of the 2011 IEEE 12th International Conference on Mobile Data Management - Volume 01*, MDM '11, pages 7–16, Washington, DC, USA, 2011. IEEE Computer Society.

[117] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. *Proceedings of the 30th VLDB Conference*, VLDB'04, pages 768–779, Toronto Canada, 2004. VLDB Endowment.

[118] Jignesh M. Patel, Yun Chen, and V. Prasad Chakka. Stripes: An efficient index for predicted trajectories. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 635–646, New York, NY, USA, 2004. ACM.

[119] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-Temporal Access Methods. *IEEE Data Eng. Bull*, pages 40–49, 2003.

[120] B. C. Ooi, K. L. Tan, and C. Yu. Fast Update and Efficient Retrieval: an Oxymoron on Moving Object Indexes. *Proc. of Int. Web GIS Workshop*, 2002.

[121] D. Kwon, S. Lee, and S. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-Tree. *Proc. MDM*, pages 113–120, 2002.

[122] Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. Indexing multi-dimensional data in a cloud system. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 591–602, New York, NY, USA, 2010. ACM.

[123] Xiangyu Zhang, Jing Ai, Zhongyuan Wang, Jiaheng Lu, and Xiaofeng Meng. An efficient multi-dimensional index for cloud data management. *Proceedings of the First International Workshop on Cloud Data Management*, CloudDB '09, pages 17–24, New York, NY, USA, 2009. ACM.

[124] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[125] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The architecture of SciDB. *SSDBM'11*, pages 1–16. Springer-Verlag, 2011.

[126] Alex Hall, Olaf Bachmann, Robert Buessow, Silviu-Ionut Ganceanu, and Marc Nunkesser. Processing a trillion cells per mouse click. *PVLDB*, 5:1436–1446, 2012.

[127] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A real-time analytical data store. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 157–168, New York, NY, USA, 2014. ACM.

[128] Apache Kylin. Apache kylin. `http://kylin.apache.org/`, 2016.

[129] S. Goil and A. Choudhary. BESS: Sparse data storage of multi-dimensional data for OLAP and data mining. Technical report, North-western University, 1997.

[130] Khalid Sayood. *Introduction to Data Compression (2Nd Ed.)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[131] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook Inc, 2007.

PEDRO EUGÊNIO ROCHA PEDREIRA

ON INDEXING HIGHLY DYNAMIC MULTIDIMENSIONAL DATASETS FOR INTERACTIVE ANALYTICS

CURITIBA

2016