Cubrick: A Scalable Distributed MOLAP Database for Fast Analytics

Pedro Eugenio Rocha Pedreira
Supervised by Luis Erpen de Bona
Federal University of Parana, Curitiba, PR, Brazil.
and Chris Crosswhite
Facebook Inc., Menlo Park, CA, USA.
pedroerp@fb.com

ABSTRACT

This paper describes the architecture and design of Cubrick, a distributed multidimensional in-memory database that enables real-time data analysis of large dynamic datasets. Cubrick has a strictly multidimensional data model composed of dimensions, dimensional hierarchies and metrics, supporting sub-second MOLAP operations such as slice and dice, roll-up and drill-down over terabytes of data. All data stored in Cubrick is chunked in every dimension and stored within containers called *bricks* in an unordered and sparse fashion, providing high data ingestion ratios and indexed access through every dimension. In this paper, we describe details about Cubrick's internal data structures, distributed model, query execution engine and a few details about the current implementation. Finally, we present some experimental results found in a first Cubrick deployment inside Facebook.

1. INTRODUCTION

Exploratory analysis of large data streams has become increasingly important as companies move toward leveraging real time data insight services. At Facebook, the ability to interactively run dashboards and analyze real time data helps to provide insights for both internal and external customers in a timely manner. However, the challenges of dynamicity, data volume and variability of queries from exploratory analysis make it challenging to fulfill users' expectations using current database technologies.

It is well known that traditional row-store databases are not suited for analytics workloads [10], and we argue that even column-stores do not perform well in highly dynamic workloads containing arbitrary queries, since data is stored in a particular (or even more than one) order. On the other hand, current OLAP databases are either (a) based on a relational database (ROLAP) [6] and therefore suffer from the same problems, (b) rely heavily on pre-aggregations [8] or (c) do not provide distributed capabilities, *i.e.*, are difficult to scale [7].

Proceedings of the VLDB 2015 PhD Workshop

In this paper, we present a new database architecture that targets Facebook's highly dynamic analytical workloads. Cubrick is an in-memory multidimensional database that can execute OLAP operations such as slice-n-dice, roll-ups and drill-down over very dynamic multidimensional datasets composed of cubes, dimensions, dimensional hierarchies and metrics as long as the cardinality of each dimension is known *a priori*. Typical use cases for Cubrick are loads of large batches or continuous streams of data for further execution of OLAP operations, generally generating small and low latency result sets to be consumed by data visualization tools. The system also supports interactive execution of queries using a subset of SQL, which we call CQL.

Data in a Cubrick cube is range partitioned (or *chunked*) in every dimension, composing a set of data containers called *bricks* where data is stored sparsely and in an unordered and append-only fashion. Cubrick works especially well in highly dynamic systems such as real time data streams; no reordering or index update is required at load time making ingested data immediately available to queries. Unlike traditional data cubes, Cubrick does not rely on any pre-calculations, rather it distributes the data and executes queries on-the-fly leveraging MPP architectures.

Queries in Cubrick can contain any type of filter, aggregations over metrics and a set of dimensions to group by. Since data is partitioned in every dimension, the system can easily prune entire bricks out based on the query's filters, so the more restrictive the search space, the faster the query is. Data is also periodically synchronized to disk using a key value store called RocksDB [1] for disaster recovery.

We have implemented Cubrick at Facebook from the ground up and deployed for a couple of pilot projects; in one of these deployments, Cubrick is able to execute queries over a dataset containing a few billion cells in hundreds of milliseconds using a single node. In addition, since Cubrick always returns ordered partial results by design, it can horizontally scale smoothly.

The remainder of this paper is organized as follows. Section 2 presents an overview of current database technologies and discusses why they are not well suited for this particular use case, while Section 2.1 points to related work. In Section 3 we detail Cubrick's internal architecture and data structures used, as well as explanations about the distributed model and how queries are executed. Section 4 shows some results we got from Facebook's current pilot implementation of Cubrick, and finally Section 5 concludes this paper.

2. BACKGROUND

Cubrick is meant to fill a gap in the current analytic databases landscape. It has been shown before [10] that traditional row based relational databases are not well suited for analytics workloads since they need to materialize entire rows instead of only the columns required by a query. Column-store databases, on the other hand, usually consume I/O and CPU more efficiently by only materializing the required columns, and since values for each column (which by definition have the same data type) are stored contiguously, they usually benefit more from compression and encoding techniques. However, since columns are stored independently these databases must provide a mechanism to correlate and materialize rows when a query requires values from more than one column.

Column-wise databases based on the C-Store [10] architecture correlate different columns by storing them using the same row based ordering, making row materialization easier since one can infer the row id based on the storage position. Despite being widely used in current data warehouse deployments, column-wise databases suffer from two major issues: (a) since data is stored according to some specific ordering, insertions, updates and deletes are usually inefficient operations (batch insertions might alleviate this issue) and (b) data pruning might not be efficient over columns other than the ones used in the sort order.

A second approach to deal with analytical workloads is to use a multidimensional model, also known as *data cubes*, modeling data in terms of dimensions, metrics and aggregations. MOLAP databases, in contrast to ROLAP systems that suffer from the previously stated problems, leverage truly multidimensional data structures but usually execute queries over pre-calculated result sets consisting of aggregations by different combinations of dimensions. These partial aggregations are usually heavily indexed and compressed so that instead of searching the entire raw data the query is executed over a much smaller set of pre-aggregated data [8].

The process of building these pre-aggregations, however, is usually computationally intensive and results in a static or hard to update cube, and can compromise its ability to scale. In addition, pre-aggregation are computed based on workload characteristics, making it unsuitable for highly dynamic workloads and ad-hoc queries such as the ones found in data exploratory analysis.

We advocate that these issues make column store and traditional MOLAP databases less suited for exploratory data analysis over dynamic datasets and that a new architecture is in order.

2.1 Related Work

In this section we provide a brief overview of the existing multidimensional database technologies able to cope with dynamic data - or the ones that do not strongly rely on pre computation.

SciDB [9] is an array database with a similar data model as implemented in Cubrick: 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 in scientific workloads, which can be quite different from regular MOLAP 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 [5] is a in-memory data cube engine that provides low query response times over spatio-temporal 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 spatio-temporal, (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 [4] 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 big. 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 footprint increase. In Cubrick, since we leverage fixed range partitioning for each dimension, the range of values a particular *brick* stores can be inferred based on its *brick_id*.

3. CUBRICK ARCHITECTURE

In this Section we provide more information about Cubrick's data model, internal data structures leveraged, distributed model and query engine.

3.1 Data Model

A Cubrick database instance CBK_i runs on a Cubrick cluster and is composed by a set of *cubes*,
$CBK_i = \{C_1, C_2, C_3, ..., C_n\}$. Each of these cubes C_j is further defined by a 3-tuple $< D_j, M_j, B_j >$.

$D_j = \{d_1, d_2, ..., d_m\}$ denotes the set of dimensions that describes the cube j, while each dimension is further described by a 4-tuple in the form
$d_d =< L_d, Card_d, Chks_d, \mathbb{M}_d >$, where

- L_d represents the set of levels for dimension d ($|L_d| > 0$). A dimension where $|L_d| > 1$ is called a *hierarchical dimension*.
- $Card_d$ denotes the cardinality of dimension d, i.e. the maximum number of distinct values allowed.
- $Chks_d = \{chk_1, chk_2, ...chk_n\}$ represent a list of disjoint sets of valid values for dimension d. In addition, $\forall i, j \in Chks_d, |i| = |j|$.
- $\mathbb{M}_i$ is a function that maps a particular value for dimension d to a chunk chk_k in Chk_i, or $M_d(v_1, v_2, ..., v_l) : Chks_d$.

Table 1: Two-dimensional sample data.

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

Complementing the cube definition, $M_i = \{m_1, m_2, ..., m_h\}$ represents the set of metrics associated with a cube i, while B_i contains a set of bricks (or buckets) that comprises the data stored on the cube, where $|B_i| = \{Chks_1 \times Chks_2 \times ... \times Chks_m\}$.

Each cell of data inserted into the cube i has the form of a tuple $Cell_i =< d_1, d_2, ..., d_n, m_1, m_2, ..., m_m >$, where d_x specify a tuple containing values for each level of a dimension x. The tuple formed by the values $< d_1, d_2, ..., d_n >$ is hereafter called *cell coordinates*. $< m_1, m_2, ..., m_z >$ define the data associated with the current coordinates, where $z = |M_i|$. Based on the coordinate values this cell is then mapped to a brick b_j in B_i by applying $\mathbb{M}_i$ to each coordinate.

3.2 Chunking and Data Structures

In a Cubrick cluster, at cube creation time the user must specify the maximum cardinality and a chunk size for each dimension, as well as the number and data types of metrics. Based on each dimension's chunk size, the cube is segmented into smaller fixed-sized cubes called *bricks*, which are used to store and locate data. Each cell stored by the cube is therefore allocated to a particular brick b depending on its coordinates. All bricks are stored in a hash table, and only created when there is data to be inserted into it.

Within a brick, cells are stored column-wise and in an unordered and sparse fashion. Each brick is composed by 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* [2]. BESS is a concatenated bit encoded buffer that represents a cell's coordinate offset inside a particular brick on each dimension. Furthermore, based on the in-brick *bess* coordinate and on the *bid*, it is possible to reconstruct a complete cell coordinate while keeping a low memory footprint. The number of bits needed to store a cell's coordinate for a particular cube i is:

$$\sum_{d=0}^{|D_i|} \lceil \lg |chk_d| \rceil$$

Optionally, a hash table can be maintained per dimension to associate labels to the actual encoded values.

Adding a new cell to a brick is just a matter of appending data to each in memory array, which is a constant time operation unless it requires re-allocation. Deletes are supported providing that their predicates only match entire bricks. Besides making Cubrick particularly well suited for applications that require high ingestion rates, the *append-only* model facilitates solving conflicts such as concurrent load operations, since the order in which cells are added to internal arrays does not matter.

Figure 1 illustrates how Cubrick organizes data from the two-dimensional two-metric sample dataset in Table 1. In

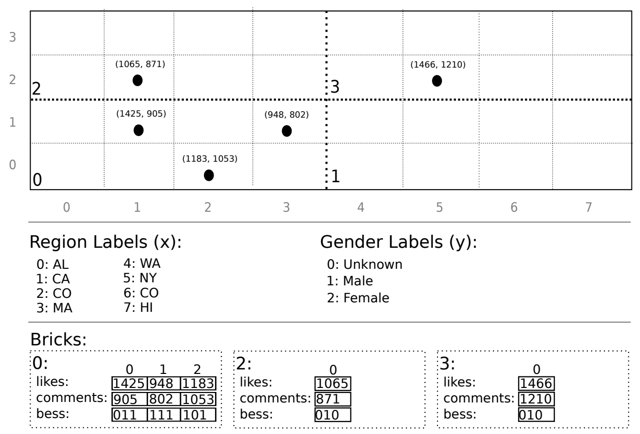

Figure 1: Cubrick data organization. From the top to the bottom: (a) an illustration of how cubrick segments the cube space into four bricks, (b) per dimension label-to-id mappings and (c) how data is organized and stored inside bricks.

order to store this data in Cubrick, one can create a cube using the following syntax:

```
CREATE CUBE cube_test
    [region 8:4 labeled, gender 4:2 labeled]
    (likes, comments);
```

This piece of DDL defines a cube named *cube_test* comprising two labeled dimensions, (a) *region* with maximum cardinality of 8 and chunk size of 4 and (b) *gender* with maximum cardinality 4 and chunk size 2. This also defines that the array will store two metrics named *likes* and *comments*. After loading the dataset, this cube will comprise by three active bricks (0, 2 and 3) containing 3, 1 and 1 cells respectively. Each brick has two metric array plus one for *bess*, which in this case takes 3 bits per cell. In this example, the total memory footprint of a cell is 67 bits (2 * 32 bits + 3 bess bits).

Lastly, another interesting feature that comes inherently from Cubrick's data organization and sparse in-brick storage is the ability to store different sets of metrics in the exact same coordinate. An appealing use case for it is, considering the same example as we shown in Table 1, if instead of the already aggregated data, the data source was composed by a stream where each row represents one like or one comment from a particular region and gender. New likes or comments can be immediatelly queriable upon appending to the correct brick (as long as the operation executed is additive). Later on, Cubrick periodically rolls-up the data (removing coordinate redundancy and aggregating the data) by a background procedure to keep the dataset small. Those roll ups are completely transparent to query execution.

3.3 Distributed Architecture

In addition to defining dimension and metrics, in a distributed cube the user must also specify the *segmentation clause*. The segmentation clause is a subset of the cube's dimensions that dictate the granularity in which data for this cube will be distributed among cluster nodes. This subset of dimensions forms larger virtual bricks (or *v_bricks*) that comprises several smaller bricks and are also numbered in row-major order. Finally, *v_bricks* are organized and assigned to cluster nodes by using consistent hash techniques — optionally *v_bricks* can be replicated by using more than one hash seed.

At query time, if the query has filters in all dimensions listed under the segmentation clause, the query request is only sent to the particular nodes that store data; otherwise, the query is broadcasted to all nodes.

3.4 Query Engine

Queries in a Cubrick cluster are highly parallelizable and composed by the following steps:

Define search space. All active bricks are evaluated in order to generate the list of local bricks that need to be scanned, based on whether they are inside, intersect or outside the search space defined by query filters. If the search space fully encompasses the brick, all cells are scanned without further action; if they intersect, each cell in the brick will first be compared against the particular filter intersecting the brick; if they are outside the search space, the entire brick is pruned. Scanning bricks that intersect a query space is more burdensome since it requires comparisons against each cell, but occur much less frequently than full scans.

Generate partial results. Queries are first distributed among cluster nodes based on a query's filters. After query distribution, on each node, bricks are assigned to worker threads that allocate dense buffers per brick based on which dimension the query groups by (the result set space) or a hash table if the buffers turn out to be too big. Once all bricks are scanned, each worker thread merges the buffers into a single partial result.

Aggregate partial results. All partial results are received and aggregated into a single buffer and returned to the user. Note that since Cubrick allocates dense buffers (or ordered maps) for partial results, merging buffers is a linear operation and can be done in parallel.

Currently Cubrick supports the *distributive* category of aggregation functions (COUNT, SUM, MIN and MAX) as defined in [3], but we can support all *algebraic* operations using the same query execution methodology.

4. EXPERIMENTAL RESULTS

In Figure 2 we show the query latency of a few Cubrick queries over two different datasets, one sized 140GB (maximum one can fit on a 144GB server) and a second one sized 1TB, both running on a 1, 10, 50 and 100 node cluster. We show the results of three different queries: (a) a query that scans the entire dataset, (b) a query containing a filter in one of the dimensions, reducing the search space to about 30% of the full dataset, and (c) a query that filters by two dimensions, where the search space is below 1% of the dataset.

5. CONCLUSIONS AND FUTURE WORK

This paper presents the architecture and design of Cubrick, a new distributed multidimensional in-memory database for real-time data analysis of large and dynamic datasets. Cubrick chunks the data in each dimension composing smaller containers called *bricks* that store data in a sparse and unordered fashion, thereby providing high data ingestion ratios.

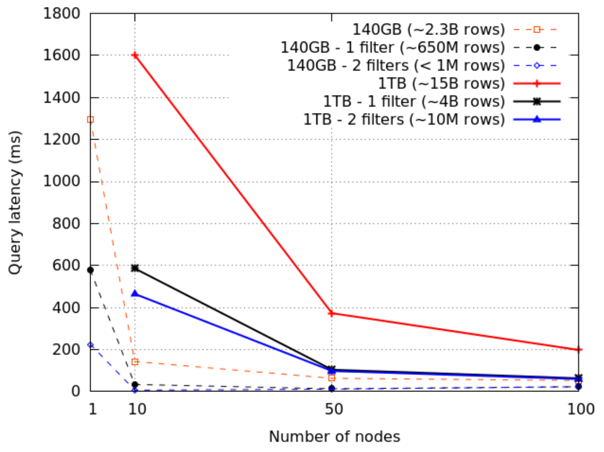

Figure 2: Query latency for different search spaces, dataset size and number of nodes.

For future work, we are exploring how to extend Cubrick in order to support: (a) efficient counting of distinct values for dimensions with very high cardinality, (b) dynamic chunking in order to support datasets with even data distribution more efficiently and (c) paginate *bricks* in and out of flash disks on the fly.

6. ACKNOWLEDGMENTS

We would like to thank the valuable feedback we got from different folks at Facebook, such as Jason Sundram, Burc Arpat, Satya Satyavarta and Miles McCrocklin, and lastly for Facebook itself for founding the project.

7. REFERENCES

[1] Facebook Inc. Rocksdb: A persistent key-value store for fast storage environments. `http://rocksdb.org/`, 2015.

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

[3] J. Gray et al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[4] A. Hall, O. Bachmann, R. Buessow, S.-I. Ganceanu, and M. Nunkesser. Processing a trillion cells per mouse click. *PVLDB*, 5:1436–1446, 2012.

[5] 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.

[6] MicroStrategy Inc. Microstrategy olap services. `https://www.microstrategy.com/us/software/products/olap-services`, 2015.

[7] Oracle Inc. Oracle essbase. `http://www.oracle.com/technetwork/middleware/essbase/overview`, 2015.

[8] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the petacube. SIGMOD'02, pages 464–475, New York, NY, USA, 2002. ACM.

[9] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of SciDB. In *SSDBM'11*, pages 1–16. Springer-Verlag, 2011.

[10] M. Stonebraker et al. C-store: A column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.