Cubrick: Indexing Millions of Records per Second for Interactive Analytics

Pedro Pedreira
Facebook Inc.
1 Hacker Way
Menlo Park, CA
pedroerp@fb.com

Chris Croswhite
Facebook Inc.
1 Hacker Way
Menlo Park, CA
chrisc@fb.com

Luis Bona
Federal University of Parana
Centro Politecnico
Curitiba, PR, Brazil
bona@inf.ufpr.br

ABSTRACT

This paper describes the architecture and design of Cubrick, a distributed multidimensional in-memory DBMS suited for interactive analytics over highly dynamic datasets. Cubrick has a strictly multidimensional data model composed of cubes, dimensions and metrics, supporting sub-second OLAP operations such as slice and dice, roll-up and drill-down over terabytes of data. All data stored in Cubrick is range partitioned by every dimension and stored within containers called *bricks* in an unordered and sparse fashion, providing high data ingestion rates and indexed access through any combination of dimensions. 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 results from a thorough experimental evaluation that leveraged datasets and queries collected from a few internal Cubrick deployments at Facebook.

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 database system 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 database 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 [19]; 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 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 databases are batch oriented systems that operate over mostly static datasets [14]. It is also well known that traditional row-store databases are not suited for analytics workloads since all columns need to be materialized when looking up a particular record [2]. We argue that even column oriented databases 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 [26]) or of keeping indexing structures up-to-date in order to efficiently answer ad-hoc queries. Moreover, current OLAP databases are either based on a relational database (ROLAP) [17] [22] and therefore suffer from the same problems, or are heavily based on pre-aggregations [24] and thus are hard to update. We advocate that a new in-memory database engine is needed in order to conform to these requirements.

In this paper we present Cubrick, a distributed in-memory multidimensional DBMS we have developed from scratch at Facebook, capable of executing indexed OLAP operations such as slice-n-dice, roll-ups and drill-down over very

[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 Section 6).

Proceedings of the VLDB Endowment, Vol. 9, No. 13

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. This new data organization technique, which we call *Granular Partitioning*, extends the notion of data partitioning as implemented by most traditional database systems.

In this paper we make the following contributions:

- We describe a novel technique used to index and organize highly dynamic multidimensional datasets called Granular Partitioning, that extends traditional database partitioning and supports indexed access through every dimension.
- We detail the basic data structures used by Cubrick to 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 incrementally 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.

The remainder of this paper is organized as follows. Section 2 describes the target workload both in terms of data and queries and how it differentiates from traditional OLAP workloads. In Section 3 we describe a novel technique that extends traditional database partitioning and can be used to organize and index multidimensional data. Further, in Section 4 we present Cubrick, its distributed architecture and details about query processing and rollups. Section 5 shows a few experiments that measure the effects of data distribution on Cubrick. Section 6 presents a comprehensive experimental analysis of our current implementation while Section 7 points to related work. Finally, Section 8 concludes this paper.

2. WORKLOADS

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 denormalized.
- **Stupidity**. The term *Stupid Analytics* was introduced by Abadi and Stonebraker [1] 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**. A common use case we have seen in the last few years is providing a UI where users can interact and execute traditional OLAP operations over a particular dataset, mostly 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 [19] 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 [8], the database system 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.

2.1 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 [14] 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 (b) we decided to handle 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 to simplify 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 4.2) such as *Male* and *Female* labels for gender dimensions or *US*, *Mexico* and *Brazil* for country dimensions. Therefore, all labels can be efficiently dictionary encoded and stored as regular integers, enabling us to build a leaner and much simpler DBMS architecture only able to operate over fixed size integers.

2.2 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 denormalization 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 4.4).

The types or filters used are predominantly exact match, but range filters are also required (in order to support date, time or age ranges, for instance). That means that indexing techniques solely based on hashing 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.

3. GRANULAR PARTITIONING

Store and index highly dynamic datasets is a challenge for any database system due to the computational costs associated with index updates. Indexing OLAP databases, in particular, is even more demanding considering that the number of records is substantially higher than in traditional transactional systems. Moreover, in realtime scenarios where a large number of load requests are constantly queued up, even small interruptions in order to update indexing structures, for instance, might stall entire pipelines.

In this section, we describe a new data organization technique that extends traditional table partitioning [5] 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.

We start in Subsection 3.1 by characterizing the current approaches that are usually leveraged to index column-oriented data and speedup queries and outline why they are not sufficient for our use cases. In Subsection 3.2 we describe the limitations that commercial DBMS offerings have regarding table partitioning support. In Subsection 3.3 we present a new approach for indexing highly dynamic datasets column-wise called *Granular Partitioning*. Finally, in Subsection 3.5 we compare the new approach proposed to traditional indexing database techniques.

3.1 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 [15]. 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 DBMS. Despite being widely adopted by OLTP DBMSs, the logarithmic overhead 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 [26] 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-Tree (*Log Structured Merge-Tree*) [20] is used to amortize the computational 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 DBMSs where the system tries to fit as much of the dataset in memory as possible to avoid burdensome disk accesses.

Lastly, database cracking [12] is an adaptive indexing technique that re-organizes the physical design as part of queries, instead of part of updates. On a DBMS leveraging this technique, each query is interpreted as a hint to crack the current physical organization into smaller pieces, whereas data not touched by any query remains unexplored and unindexed. This technique has the advantage of adapting the physical design according to changes in the workload and without hu-

man 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.

All three techniques described above have a inherent overhead for insertions, which prevents their ability to scale under highly dynamic workloads

3.2 Database Partitioning

Partitioning is an umbrella term used in database systems 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 interfering with the others. Partitioning can also be leveraged to prune data at query time by entirely skipping partitions that do not match query filters.

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, partitioning on most traditional DBMSs is one-dimensional (even though multiple columns can be part of the key) and have a low limit on the total number of partitions allowed per table – usually about a few thousands [21] [18] — 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.

3.3 A New Approach

In this paper, we propose a new technique that extends traditional database partitioning and can be leveraged to index highly dynamic multidimensional datasets. We extend the notion of partitioning by assuming that all tables in the system are range partitioned by every dimension column, and that the cardinality and range size of each dimension are known beforehand. We also assume that all string values are dictionary encoded and internally represented as monotonically increasing integers.

Considering that tables can have hundreds of dimensions, and depending on the cardinality and range size defined for each of these columns, the database system needs to be able to efficiently handle tens to hundreds of millions of small partitions. As the number of active partitions scale (partitions containing at least one record), a few problems arise: (a) how to 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.

Finding the correct partition to insert a new record. Every time a new record enters the database system for ingestion, the partition to which it should be appended must be found based on the values of the dimension columns. Since the cardinality and range size of each dimension are known beforehand, a row-major numbering function can be used to uniquely assign ids to every possible partition. Therefore, based on a input record the target 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 functions can be leveraged to map the multidimensional partition map into a one dimensional space (the partition id), such as Z-ordering, space filling curves and Hilbert curves. We decided to use a row-major function because of its simplicity and low overhead calculation. Thus, 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:

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

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. Hence, 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 vectors 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 1: Two-dimensional example dataset.

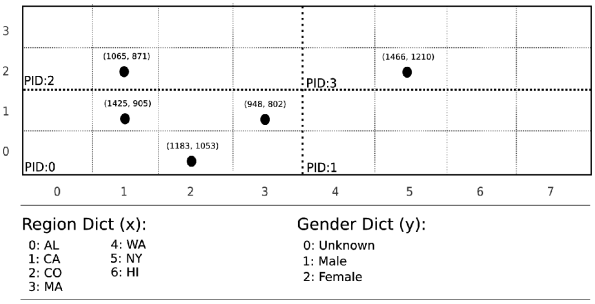

Figure 1: Granular Partitioning data layout. Illustration of how records are associated with partitions on the two-dimensional example shown in Table 1 and per dimension dictionary encodings.

Figure 1 illustrates how data is distributed between partitions given the example two dimensional input dataset shown in Table 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.

Scanning the correct partitions at query time. Since the dataset is partitioned by every dimension column, a multidimensional tree-like data structure would be required in order to provide sublinear searches by the partitions that match a particular set of filters. However, according to experimental results in some of our datasets (not shown in this paper), given the number of partitions and dimensions we are required to store, the computational requirements of maintaining and iterating over multidimensional data structures such as R-Trees [10] and K-D-B Trees [6] exceeds the computational costs of testing every active partition against the filters of a query.

Besides, testing every active partition can be efficiently implemented considering that (a) the list of active *pid*s can be stored sequentially in memory and take advantage of memory locality and cache friendly iterations and (b) 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 order to reduce the amount of indexing metadata maintained per partition the following equation can be leveraged to infer the offset on dimension d solely based on a *pid*:

$$offset_d = pid \times \left(\prod_{k=1}^{d} \frac{D_k}{R_k} \right)^{-1} \bmod \frac{D_d}{R_d} \qquad (2)$$

Hence, by only storing a single 64 bit integer per partition of indexing metadata it is possible to decide whether a partition matches a set of filters and therefore needs to be scanned, or can be safely skipped.

Efficiently store millions of partition. 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 DBMS 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 [7]) to provide disaster recovery capabilities, and persistency can be achieved through replication.

3.4 Extending Dimensions

In order to minimize the amount of re-organization required when a dimension needs to be extended (for instance, in cases where the estimated cardinality is exceeded), one possible strategy is to re-calculate the ids of every active partition. Even though this approach only requires re-numbering active partitions in memory, it can impact the distributed model (as described in Section 4.3) and cause data to be re-flushed to disk if data placement is determined by the partition id. A second strategy is to row-major number the space extension on its own, starting from the total number of partitions the table had before the extension.

	Query	**Insertion**	**Space**
Naive	$C_T * N_T$	C_T	$C_T * N_T$
Row-store (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 Partitioning	$C_Q * N_Q * \delta$	C_T	$C_T * N_T$

Table 2: Comparison between different indexing methods.

3.5 Comparison

Table 2 presents a model that can be used to compare the worst-case complexity and trade-offs between different 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 time, or the time required to update the indexing data structures every time a new record is ingested, and space complexity, or the space required to maintain the indexing structures.

We present and compare four hypothetical DBMSs implementing different indexing techniques using our model: (a) a *naive* row-store DBMS that does not have any indexing capabilities, but rather only appends new records to the end of its storage container, (b) a row-store DBMS maintaining one B+Tree based index per column, (c) a column-store DBMS based on the C-STORE architecture that keeps one projection ordered by each column, and (d) a DBMS implementing the Granular Partitioning technique presented in Section 3.3.

Since the naive DBMS (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 DBMS 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 DBMS 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 DBMS 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 DBMS 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 maintained.

4. CUBRICK ARCHITECTURE

Cubrick is a novel distributed in-memory OLAP DBMS specially suited for low-latency analytic queries over highly dynamic datasets. In order to operate at our scale, rather than relaxing guarantees and removing pieces from traditional database systems, 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.

As described in Section 2, the input for Cubrick are denormalized 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* [9]). 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 the Granular Partitioning technique described in Section 3. 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.

4.1 Terminology

In this subsection, we define the terminology we use to describe Cubrick throughout the paper. 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.

4.2 Data Organization

Cubrick organizes and stores all data in-memory inside partitions referred 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).

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* [9]. 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* [23]. Internally, all data is stored based on these ids and only converted back to strings before returning data to users.

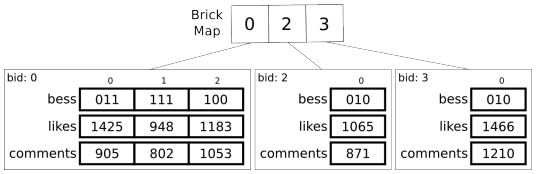

Figure 2: Cubrick internal data organization for the dataset presented in Table 1.

Figure 2 illustrates how Cubrick organizes the dataset shown in Table 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.

4.3 Distributed Architecture

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 hash ring [13] and data is distributed among the nodes by hashing each brick id into this ring and thereby assigning bricks to nodes. Queries are always forwarded to all nodes that store data for a cube and their results are aggregated before returning the result set to the client. Lastly, replication is done 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.

4.4 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 [4], 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 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 cardinality 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 3 shows a comparison of the query execution times using dense and sparse 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

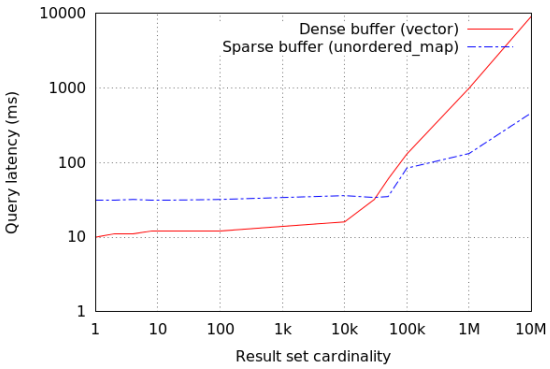

Figure 3: Query latency for different result set cardinalities using sparse and dense buffers.

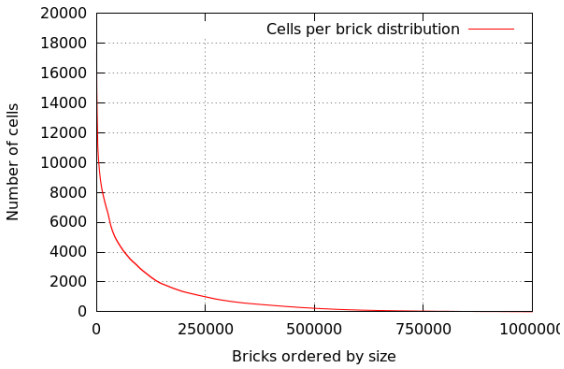

Figure 4: Distribution of cells per brick in a real 15-dimensional dataset.

they are available, until all tasks are finished and the full result set is generated. Lastly, the ids used internally are translated to labels, formatted according to the output requested and sent back to the client.

4.5 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 3 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 3: 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. 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.

To illustrate the distribution of a real dataset, Figure 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

# 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 4: Full table scan times for a 1 billion cells dataset following a perfectly even and skewed distribution (as shown in Figure 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.

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 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 4. 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 4. 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.

6. EXPERIMENTAL RESULTS

In this Section, we present an experimental evaluation of our current implementation of Cubrick. For all experiments, the servers used have 32 logical CPUs and 256 GB of memory, although we have used different cluster sizes for different experiments. 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.

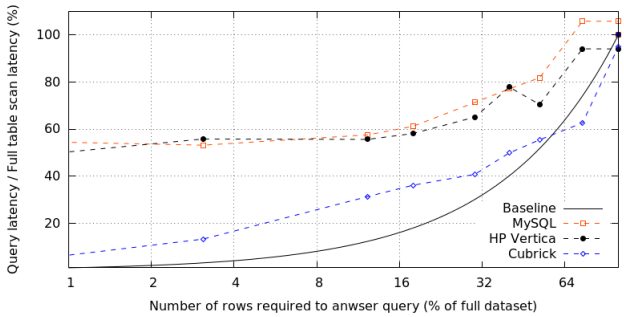

Figure 5: Relative latency of filtered OLAP queries compared to a full table scan.

Figure 5 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 DBMS 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 DBMS.

6.2 Queries

In this Subsection, 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.

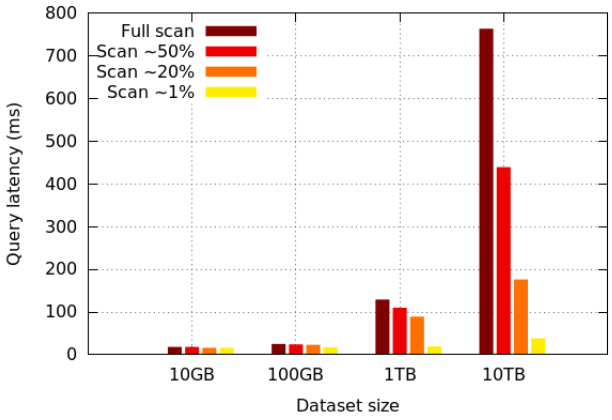

Figure 6: Queries over different dataset sizes on a 72 node cluster. The first query is non-filtered (full scan) and the following ones contain filters that match only 50%, 20% and 1% of the dataset.

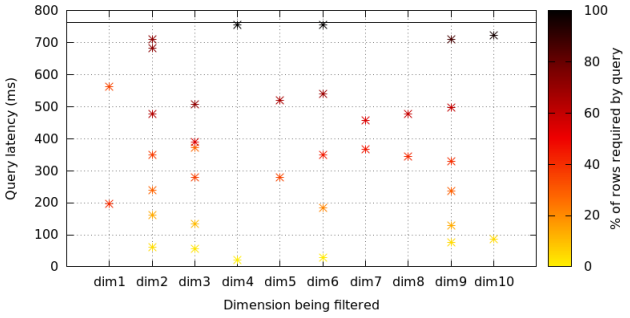

Figure 7: Latency of queries being filtered by different dimensions over a 10TB dataset on a 72 node cluster.

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 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.

In the second 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 7 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 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 Subsection, 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.

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 8 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.

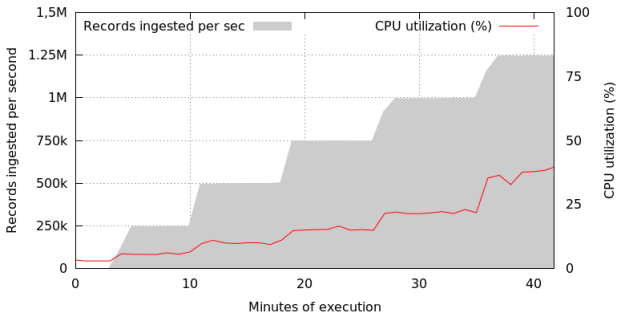

Figure 8: Single node cluster CPU utilization when ingesting streams with different volumes.

6.3.2 *Rollups*

As described in Section 4.5, 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. 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 9: Memory utilization of servers ingesting data in real time with rollup enabled.

Figure 9 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 metric with no 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.

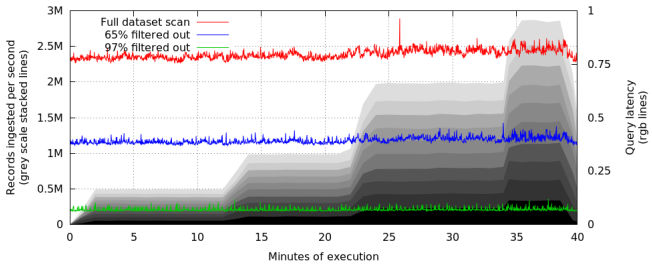

Figure 10: Latency of queries on a 10 node cluster ingesting streams of different volumes.

Figure 10 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 Subsection 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.

7. PREVIOUS WORK

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

SciDB [25] is an array database with a similar data model. 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 [16] is a in-memory data cube engine that provides low query response times over spatiotemporal multidimensional datasets. 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 spatiotemporal,

(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 [11] 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 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*.

Lastly, Scuba [3] is another in-memory data analysis tool that relies on partitioning, but only indexes partitions based on a single dimension - timestamp.

8. CONCLUSIONS

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

For future work, we intend to continue exploring new strategies to make Cubrick more generic and better support less curated datasets. Things we plan to tackle are (a) automated schema generation, or improve our dimension cardinality estimation and (b) more efficient storage for datasets with very skewed data distribution.

9. REFERENCES

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

[2] D. J. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *SIGMOD*, Vancouver, Canada, 2008.

[3] L. Abraham et al. Scuba: Diving into data at facebook. *Proc. VLDB Endow.*, 6(11):1057–1067, Aug. 2013.

[4] A. Agarwal, M. Slee, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook Inc., 2007.

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

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

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

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

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

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

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

[12] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research*, pages 68–78, Asilomar, CA, USA, 2007.

[13] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, pages 654–663, New York, NY, USA, 1997. ACM.

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

[15] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *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.

[16] 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.

[17] MicroStrategy Inc. Microstrategy olap services. `https://www.microstrategy.com/us/software/products/olap-services`, 2016.

[18] MySql. Mysql 5.7 reference manual. `http://dev.mysql.com/doc/refman/5.7/en/`, 2016.

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

[20] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[21] Oracle Inc. Oracle database 12c. `http://www.oracle.com/technetwork/database/enterprise-edition/documentation/`, 2016.

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

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

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

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

[26] 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.