A Distributed Interactive Cube Exploration System

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Prasanth Jayachandran

Graduate Program in Computer Science and Engineering

The Ohio State University

2013

Master’s Examination Committee:

Dr. Arnab Nandi, Advisor

Dr. Srinivasan Parthasarathy
ABSTRACT

As the amount of data generated from web, mobile and social media increases rapidly, analytics using OLAP (Online Analytical Processing) data cubes is getting increasingly popular among organizations. In a typical scenario, this analysis is performed using BI tools to quickly get insights from the pre-materialized multi-dimensional aggregated data. We introduce DICE, a distributed interactive system that uses a novel session-oriented model for online data cube exploration, which is designed to provide the user with interactive sub-second latencies for specified accuracy levels. We provide a novel framework that combines three concepts: faceted exploration of data cubes, speculative execution of queries and query execution over sampled data. Our system uses a combination of intuitive frontend for faceted cube exploration and distributed query execution backend that guarantees interactive latencies. We catalog the challenges encountered in building such a system, we discuss design considerations, implementation details and optimizations of our system. Experiments demonstrate that cube exploration using DICE at billion-tuple scale is at least 33% faster than current approaches. As shown in our video demonstration, DICE allows the user to fluidly interact with billion-tuple datasets while maintaining sub-second interactive response times.
This document is dedicated to my family and friends.
ACKNOWLEDGMENTS

I would like to convey my deepest gratitude to Dr. Arnab Nandi for giving me the opportunity to work on a very interesting research area. I would also like to acknowledge and thank him for being my academic advisor and for providing guidance throughout my research. From the day I joined his team (which also happened to be the first day of his tenure at OSU) he gave me challenging research problems to work with that not only stretched my abilities to think but also introduced me to a wide gamut of knowledge which I was never exposed to. The problems and standards set by him in research were of very high quality, which always resulted in a positive outcome. He also gave me the complete freedom to explore and learn on my own way. I would also like to thank him for providing opportunities that resulted in open source contributions, which had (and will have) profound influence in my career.

I would also like to thank Dr. Srinivasan Parthasarathy for being my thesis committee member, for providing valuable suggestions and constructive criticism for this work.

I would like to acknowledge and thank Karthik Tunga and Niranjan Kamat with whom I spent most of the time in lab working and understanding the intricacies of the research problem. I still remember the days when we used to sit late night in the lounge and shout; “nailed it” after finding solutions to complex issues. I also feel that this is the right blend of people to work with, one with strong theoretical background and another with strong
systems background, which eventually yielded a very good outcome. This work would not have been possible without their effort, guidance and support. I would also like to thank all members of the Datateam for reviewing papers and providing valuable suggestions before conference submissions and also for supporting our work.

I would like to thank all my friends at OSU especially Nithin, Pundi, Yogi, Green Card, Ravi, Ajith Dixit, Sirji, Susa, Ananth, Rohit, Boss, Arun, Big Mike, Ramya, Chatty, Tarun, Farhang, Manas, Nair, Sabai, Tonty, Thalaivare and everyone for providing all the fun moments and for making OSU journey memorable.

Special thanks to my special friends at Samsung Tamil Sangam, members of Xcary and Smiley Faction, members of Rockers Club who always kept smile on my face. I can never forget the time that I spent with you guys and I will always cherish those happy and wonderful moments. I would like to specially thank my close friends Jimmy, Aatha, Soumya, Baba who stood by me through all my happiness and trouble and provided me with invaluable guidance and emotional support.

I don’t have enough words to thank my parents, for their love, affection, for the innumerable sacrifices they made for me and for the confidence they had in me which always kept me moving forward in life. I can’t express through words for the love, affection, support, confidence and motivation, my sister gave me throughout my life. I would also like to thank athan, all my cousins and relatives who always cared and wished
for my well being. Finally, Akhil kutty for all his naughtiness, cute little smile, and his addiction to play kicking ball and catching ball with me, which made me forget everything in the world. I am always indebted to all of their love and care.
VITA

2002 to 2004 ...........................................BVB Higher Secondary School, Erode, Tamil Nadu, India

2004 to 2008 ...........................................B.E(Hons), Computer Science, Birla Institute of Technology and Science, Pilani, India

January to June 2008 .........................Intern, SAP Labs, Bangalore, India

July 2008 to August 2011 ......................Software Engineer, Samsung Software R&D Center, Noida, India

September 2011 to present ....................Graduate Teaching Associate, Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA

PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer Science and Engineering
# TABLE OF CONTENTS

ABSTRACT ................................................................................................................................. ii

ACKNOWLEDGEMENTS ........................................................................................................ iv

VITA ................................................................................................................................................ vii

PUBLICATIONS ....................................................................................................................... vii

FIELDS OF STUDY ................................................................................................................... vii

TABLE OF CONTENTS ............................................................................................................. viii

LIST OF TABLES ....................................................................................................................... xii

LIST OF FIGURES ..................................................................................................................... xiii

LIST OF ALGORITHMS ............................................................................................................ xiv

Chapter 1: Introduction ............................................................................................................. 1

1.1 Terminologies ................................................................................................................... 3

1.1.1 Data Cube .................................................................................................................. 3

1.1.2 Dimensions ............................................................................................................... 3

1.1.3 Measure .................................................................................................................... 3

1.1.4 Cube Lattice .............................................................................................................. 4

1.1.5 Drill down .................................................................................................................. 5

1.1.6 Rollup ......................................................................................................................... 5

viii
1.2 Contributions.................................................................................................................. 6
1.3 Organization .................................................................................................................... 8

Chapter 2: Offline Cube Materialization .............................................................................. 10

2.1 Apache Pig....................................................................................................................... 10

2.2 Naïve Cubing in Apache Pig .......................................................................................... 11
  2.2.1 Using Union of Group-by’s ...................................................................................... 11
  2.2.2 Using User Defined Function .................................................................................. 12
  2.2.3 Using CUBE Operator in Apache Pig ....................................................................... 14

2.3 MR-Cube for Holistic Measures in Apache Pig ............................................................. 19
  2.3.1 Lack of Statistics ...................................................................................................... 20
  2.3.2 Low Overhead Sample Loader ............................................................................... 21

Chapter 3: Online Cube Materialization .............................................................................. 26

3.1 Motivating Example ....................................................................................................... 28

3.2 Common Approaches ..................................................................................................... 30

3.3 Data Model and Preliminaries ....................................................................................... 31
  3.3.1 Faceted Exploration of Data Cubes .......................................................................... 31

3.4 Distributed Execution ..................................................................................................... 37

3.5 Querying over Sampled Data ........................................................................................ 37

3.6 Sampling and Result Quality ....................................................................................... 41

3.7 The DICE System ......................................................................................................... 43
  3.7.1 Speculating Queries in a Session ............................................................................. 43
  3.7.2 System Architecture ............................................................................................... 45
Chapter 5: Related Work .................................................................................................................. 81

Chapter 6: Conclusion and Future Work ......................................................................................... 85

References ....................................................................................................................................... 87

Appendix A: Combination of Variances.......................................................................................... 90
LIST OF TABLES

Table 1: List of notations used in this section ........................................................... 34
Table 2: List of notations used in this subsection ...................................................... 38
LIST OF FIGURES

Figure 1: Cube lattice for 3 non-hierarchical dimensions ........................................... 5
Figure 2: MapReduce job sequence for MR-Cube algorithm ........................................ 25
Figure 3: DICE Problem Setting .................................................................................. 28
Figure 4: Cube lattice for hierarchical attributes ......................................................... 32
Figure 5: Faceted cube exploration traversals .............................................................. 33
Figure 6: Threaded Master-only DICE Architecture (Approach 1).................................. 47
Figure 7: Threaded Master-Slave DICE Architecture (Approach 2)................................. 48
Figure 8: Evented Master-Slave DICE Architecture (Approach 3)................................. 50
Figure 9: Varying Size of Dataset: CLUSTERSMALL .................................................. 64
Figure 10: Varying Size of Dataset: CLUSTERCLOUD ............................................... 64
Figure 11: Accuracy over a workload ............................................................................ 66
Figure 12: Impact of Number of Dimensions ............................................................... 67
Figure 13: Varying the Number of Slave Nodes ............................................................ 68
Figure 14: Cache Hit Change with Sampling Rate Change ........................................... 69
Figure 15: Individual Latencies for Anecdotal Query Session: NOSPEC, DICE and
Perfect Algorithms ....................................................................................................... 70
Figure 16: Impact of Various Techniques ....................................................................... 71
Figure 17: Varying the Table Shard Size ....................................................................... 72
Figure 18: Snapshot of DICE Frontend ......................................................................... 75
Figure 19: Query Flow ................................................................................................... 76
LIST OF ALGORITHMS

Algorithm 1: Core Exploration Loop ................................................................. 55
Algorithm 2: DICE Execution Strategy ............................................................. 56
Algorithm 3: Facet Selection ........................................................................... 57
Chapter 1: Introduction

Recent advancements in web and mobile technologies have tremendously increased the rate at which data is generated. It is estimated that hundreds to few thousands of terabytes of structured and unstructured data is being generated from social networking sites per day. Hidden in this vast amount of data is invaluable information, which can be analyzed/mined for business intelligence, sentiment analysis, product usage analysis etc. Large-scale analytics has found a growing number of use cases in a variety of disciplines, from business to the sciences, from healthcare to genome sequencing, from financial services to digital media services. With the rapid rise in data, and the reliance of data-driven insights for decision-making, planning and analysis, the role of analytics over massive datasets has become a critical one.

With proliferation of large-scale data infrastructure, it is not uncommon for end-users to expect direct fine-grained control over large amounts of data. The availability of both dedicated and dynamically provisioned distributed computational resources allow analyses that were typically handled by database administrators to be performed by the end-users of the analysis themselves. Further, there is increasing demand in real-time or near real-time analytics, where all analysis is performed on in-situ data, such as constantly updating logs that are being appended to in a batched manner. As detailed in
In the following paragraphs, despite the availability of performant, distributed and scalable infrastructure, there exist several challenges to large-scale analytics.

In addition to the typical use cases of reporting where predetermined query templates are run over batches of new incoming data, and mining where data is analyzed to discover interesting patterns of information, there has been a sharp rise in the demand for ad-hoc analytics, exposed to the user over interfaces for business intelligence, interactive dashboards and advanced domain-specific data-driven applications. These challenges are exacerbated in the scope of ad-hoc analysis over a cube representation of the data [12], where a CUBE can be considered as the union of aggregations across all combinations of dimensions of the dataset, for a given measure. Such a representation is useful for the purpose of exploratory data analysis, since successive investigatory questions can be answered in the form of drilldown or rollup queries. However, given the exponential space of possible aggregations, traditional approaches such as pre-materialization do not suffice.

Given the increasing need for large-scale data analysis, low-cost availability of large-scale data infrastructure, the need for real-time or near real-time analytics and lack of tools for exploratory analysis over data cube representation, we were highly motivated to “build a distributed and interactive system for exploratory data cube analysis that executes ad-hoc aggregate queries over billions of rows under sub-second latency.” which forms the basis for our research.

Following section describes the terminologies that are used throughout this paper.
1.1 Terminologies

1.1.1 Data Cube

In Online Analytical Processing Systems (OLAP), a Data Cube [12] is way of organizing and analyzing multi-dimensional data over some measure of interest.

1.1.2 Dimensions

Dimensions are the attributes of the data that are used for analysis. Dimensions may or may not be organized in hierarchies. For example, time dimension can be organized with three levels of hierarchies (year, month, week).

1.1.3 Measure

Measure is a term for numerical facts that are analyzed against various dimensions of data. Based on the type of aggregation function used, measures can be classified as algebraic and holistic measure.

1.1.3.1 Algebraic Measures

Algebraic measures are those in which higher-level aggregates can be computed from lower level aggregates. Algebraic measures are also distributive in the sense that they can be computed independently and aggregated together using a final function to get the overall result. Measures like COUNT, SUM, MAX etc. are examples of algebraic measures with final functions SUM, SUM, MAX respectively. Some measures are semi-
distributive like AVERAGE that can be re-written to COUNT, SUM distributive measures to compute the overall average in distributed manner.

1.1.3.2 Holistic Measures

Holistic measures on the other hand are not distributive and so computing the higher level aggregates from lower-level aggregates is hard. Measures like COUNT + DISTINCT, TOP-K, MEDIAN etc. are holistic. Section 2.3 describes techniques to tackle the challenges with computing holistic measures in a distributed manner.

1.1.4 Cube Lattice

Cube lattice is spatial representation of all possible groupings (from lowest to highest) of dimensions. Each node in the lattice is referred as a cube region and the actual set of values belonging to each cube region is referred as a cube group. Figure 1 is the cube lattice for 3 non-hierarchical dimensions (zone, month, topic) with the lowest aggregation level at the bottom most level of the lattice and highest aggregation level at the top most level of the lattice. The nodes in the lattice for example (zone, *, *), (*, month, topic) etc. are the cube regions and the set of domain values corresponding to these regions (“east”, *, *) , (*, “april”, “iphone”) respectively are the cube groups.
1.1.5 Drill down

Drill down is navigational pattern in cube lattice where the user moves from a higher level of aggregation to a lower level. In the above lattice navigating from cube region (zone,*,topic) to (zone,month,topic) is a drill down operation.

1.1.6 Rollup

Rollup is opposite of drill down where the user moves from a lower level of aggregation to a higher level of aggregation. A navigation from cube region (zone,*,topic) to (zone,*,*) in the above lattice can be considered as a rollup operation.
1.2 Contributions

1.2.1 Offline Cube Materialization Support In Apache Pig

Support for cube operation in Apache Pig [39] was done as a part of Google Summer of Code, 2012 (GSoC’12). The implementation contained two different phases.

- *Naïve cubing implementation* – The first phase involved implementation of CUBE as a native operator in Pig. Adding operator to Pig involved changes to Pig Latin [37] grammar files and changes to logical plan layer. The grammar files changes were related to the syntax of CUBE operators whereas the changes to logical plan layer involved rewriting the CUBE operator to projection expression with cube/rollup user defined function. Although naïve implementation worked well for algebraic measure, it failed for holistic measures at a large scale. This resulted in the implementation of MR-Cube algorithm [24] to make holistic aggregation highly scalable.

- *Distributed cubing for holistic measures* – The second phase primarily involved implementation of MR-Cube algorithm, which required modifications to physical and map-reduce layer of query plan. The core implementation involved rewriting query plans to accommodate statistics gathering, sampling and post-aggregation jobs with cube materialization job when encountered a holistic measure.
1.2.2 System For Online Cube Exploration

This forms the core of the investigative research component which involved building a distributed system for interactive and online cube exploration.

- **Distributed Execution Engine** – We developed a distributed query execution engine that guarantees interactive sub-second latencies for query execution over a user-defined sample of data. The execution engine was implemented using the actor model, which apart from providing the above guarantees also provides high concurrency, high scalability and high throughput with low latency. The execution engine also supports bounded time execution of speculative queries.

- **Faceted Cube Exploration Model** – We introduce a novel session-oriented cube exploration model. We introduce cube exploration in the context of facets, which are the basic units of cube explorations. We model the transitions from one facet to another using a set of traversals.

- **Speculative and Bounded Time Execution** – We introduce speculative execution of queries based on the currently executing query. Based on the current facet in the cube lattice we enumerate all possible transitions, prioritize and execute them within a fixed bounded time (5000ms in our implementation). We also cache the results of the speculated queries so that the subsequent queries are served from the cache in case of successful speculation.
1.2.3 Frontend For Online Cube Exploration System

To explore the full potential of our online cube exploration system, we built an easy-to-use and interactive frontend that provides an intuitive traversal based cube exploration for novice users and manual exploration using query editor for advanced users. The frontend supports two different modes – online mode for interactive cube exploration and offline mode for batch-oriented offline cube materialization.

1.3 Organization

The rest of the thesis is organized as follows: in Chapter 2 we begin with offline cube materialization support that we provided for Apache Pig which also details the two phases of cubing implementation. The challenges and techniques used to tackle them are discussed in detail for naïve and MR-Cube implementation. Chapter 3\(^1\) discusses in detail the various components of our distributed and interactive online cube exploration system. This section also introduces our novel faceted cube exploration model along with speculative execution and caching of queries. Various experiments, evaluations and optimizations of our system are also discussed in this section. In Chapter 4\(^2\) we discuss

---

\(^1\) This section is adapted from our full paper conference submission (refer publications section) with some sections explained in detail than original submission.

\(^2\) This section is adapted from our demonstration paper conference submission (refer publications section) with some sections explained in detail than original submission.
various features and usage of our frontend for online cube exploration system. Finally, Chapter 5 and Chapter 6 cover the related work, conclusion and future work.
Chapter 2: Offline Cube Materialization

Offline cube materialization is a process of fully or partially materializing the data cube over the set of user specified dimensions and storing the materialized results in a table/file. This materialized view can be refreshed or recomputed later in case if there are updates to input data. The advantage of this pre-materialization is that all future ad-hoc queries can be answered by simply looking up the materialized result/view.

As a part of GSoC’12, we provided support for offline cube materialization in Apache Pig [39]. Following sections will give a brief introduction to Apache Pig, the challenges we faced while implementing the CUBE operator and the techniques we used to overcome those challenges. It also discusses the implementation details of MR-Cube algorithm [24] for highly scalable distributed cubing for holistic measures.

2.1 Apache Pig

Apache Pig is a software platform for large-scale data analysis that runs on top of Apache Hadoop [40]. Apache Hadoop provides a reliable, fault tolerant, scalable and distributed computing framework for processing large datasets running across cluster of commodity servers. Applications that run on Hadoop are typically written using MapReduce [41],
which is a software-programming paradigm for writing massively parallel applications to process multi-terabytes of data on hundreds and thousands of commodity machines. Apache Pig provides a high-level dataflow language called Pig Latin, which is used to write data flow sequences and transformations in a simple and readable manner. These Pig Latin scripts are compiled to an optimized sequence of MapReduce jobs, which are then executed on a Hadoop cluster. Following section discusses the challenges and implementation details of CUBE operator in Apache Pig.

2.2 Naïve Cubing in Apache Pig

Apache Pig did not have support for the CUBE operator natively until 0.11.0 version. There are two different ways in which CUBE operation can be achieved before 0.11.0 version. We analyze both ways with a walkthrough example to get a better picture. For simplicity let us perform cubing over 2 dimensions (region and month) and compute total revenue using SUM as measure.

2.2.1 Using Union of Group-by's

One way to achieve the above CUBE operation is by using union of multiple group-bys statements with combinations of all possible aggregations. As it can be seen from the following pig latin script below, this method is extremely verbose even for small number of dimensions. As number of dimensions increases there is exponential increase in the number of group-by statements which will make the program hard to read and maintain.
inp = LOAD '/path/to/input/file' USING PigStorage('t') as (region:chararray, month:chararray, revenue: long);
grp_rm = GROUP inp BY (region, month);
grp_r = GROUP inp BY region;
grp_m = GROUP inp BY month;
grp_all = GROUP inp ALL;
flt_rm = FOREACH grp_rm GENERATE FLATTEN(group), SUM(inp.revenue) as total_revenue;
flt_r = FOREACH grp_r GENERATE FLATTEN(group), null, SUM(inp.revenue) as total_revenue;
flt_m = FOREACH grp_m GENERATE null, FLATTEN(group), SUM(inp.revenue) as total_revenue;
flt_all = FOREACH grp_all GENERATE null, null, SUM(inp.revenue) as total_revenue;
result = UNION flt_rm, flt_r, flt_m, flt_all;
STORE result INTO '/path/to/output/file';

2.2.2 Using User Defined Function

Another way to achieve CUBE operation is to write User Defined Function (UDF) that takes a tuple as input and outputs a databag with all possible aggregations corresponding
to that tuple. Similar UDF can also be written for performing ROLLUP operation. The following pig latin script illustrates the usage of CubeDimensions UDF

```pig
inp = LOAD '/path/to/input/file' USING PigStorage('t') as (region:chararray, month:chararray, revenue: long);
result = FOREACH inp GENERATE FLATTEN(CubeDimensions(region,month)) as (region,month), SUM(inp.revenue) as total_revenue;
STORE result INTO '/path/to/output/file';
```

Though this method is much less verbose that the first method, it will lead to unwieldy code when there are multiple combinations of CubeDimensions and RollupDimensions and also this method doesn’t provide opportunities for cube specific optimizations.

We improvised this method in first phase of GSoC’12 by adding syntactic sugar over the UDFs mentioned above. This not only made syntax for CUBE operation clear and concise but also paved way for adding more CUBE specific optimizations, which forms the second phase of our GSoC’12 implementation. First phase implementation is a naïve implementation focused on providing support for CUBE operators natively in Pig and second phase focused on a highly scalable distributed cubing implementation for holistic measures. In the naïve implementation, the CUBE queries are rewritten internally at the logical plan layer to projection expression (using foreach operator) with CubeDimensions or RollupDimensions UDF. The CUBE operator thus acts as a syntactic sugar for

---

3 This UDF was available in 0.10.0 version of Pig.
https://issues.apache.org/jira/browse/PIG-2168
CubeDimensions and RollupDimensions UDF. The syntax for the new CUBE operator and examples of usage are described in the following section.

2.2.3 Using CUBE Operator in Apache Pig

2.2.3.1 Cube Operation

For a specified set of dimensions, a cube operation is as per [12] which computes aggregates all possible combinations of group by dimensions. For example, CUBE(product, location) will generate \{(product, location), (product, null), (null, location), (null, null)\} combinations for all tuples in product and location dimensions, where (null, null) represents the grand aggregate total. The number of group-by combinations generated by cube for \( n \) dimensions will be \( 2^n \).

2.2.3.2 Rollup operation

For a specified set of dimensions, a rollup operation is as per [12] which computes multiple levels of aggregates based on hierarchical ordering. For example, ROLLUP(region, state, city) will generate \{(region, state, city), (region, state, null), (region, null, null), (null, null, null)\} combinations for all tuples in region, state and city dimensions. It computes aggregates from the most-detailed level of aggregation (region, state, city) to the grand total (null, null, null). Rollup is useful when there is hierarchical ordering in the dimensions. The number of group by combinations generated by rollup for \( n \) dimensions will be \( n + 1 \).
2.2.3.3 CUBE Operator Syntax

```plaintext
outalias = CUBE inalias BY { CUBE expression | ROLLUP expression }, [ CUBE expression | ROLLUP expression ] [PARALLEL n];
```

`outalias` - The name of the output relation

`inalias` - The name of the input relation

*CUBE*, *BY*, *ROLLUP* - keywords

`expression` - projections (dimensions) of the relation.

`PARALLEL n` - Increase the parallelism of a job by specifying the number of reduce tasks, n. The default value for n is 1 (one reduce task).

2.2.3.4 Example of CUBE operation

```plaintext
salesinp = LOAD '/pig/data/salesdata' USING PigStorage(',',) as (product:chararray, year:int, region:chararray, state:chararray, city:chararray, sales:long);
cubedinp = CUBE salesinp BY CUBE(product,year);
result = FOREACH cubedinp GENERATE FLATTEN(group), SUM(cube.sales) as totalsales;
```

2.2.3.4.1 Output

For a sample input tuple (car, 2012, midwest, ohio, columbus, 4000), the above query with cube operation will output

(car,2012,4000)
(car,,4000)
2.2.3.4.2 Output Schema

grunt> describe cubedinp;

Note the second column, ‘cube’ which is a bag of all tuples that belong to ‘group’. Also note that the measure attribute ‘sales’ along with other unused dimensions in load statement are pushed down so that it can be referenced later while computing aggregates, like in this case SUM(cube.sales).

2.2.3.5 Example ROLLUP operation

salesinp = LOAD '/pig/data/salesdata' USING PigStorage('"') as (product:chararray, year:int, region:chararray, state:chararray, city:chararray, sales:long);
rolledup = CUBE salesinp BY ROLLUP(region,state,city);
result = FOREACH rolledup GENERATE FLATTEN(group), SUM(cube.sales) as totalsales;

2.2.3.5.1 Output

For a sample input tuple (car, 2012, midwest, ohio, columbus, 4000), the above query with rollup operation will output
2.2.3.5.2 Output Schema

grunt> describe rolledup;


2.2.3.6 Example of CUBE and ROLLUP operation combined

If CUBE and ROLLUP operations are used together, the output groups will be the cross product of all groups generated by cube and rollup operation. If there are \(m\) dimensions in cube operations and \(n\) dimensions in rollup operation then overall number of combinations will be \(2^m \times (n + 1)\).

```
salesinp = LOAD '/pig/data/salesdata' USING PigStorage('','') as (product:chararray, year:int, region:chararray, state:chararray, city:chararray, sales:long);
cubed_and_rolled = CUBE salesinp BY CUBE(product,year), ROLLUP(region, state, city);
```
result = FOREACH cubed_and_rolled GENERATE FLATTEN(group),
SUM(cube.sales) as totalsales;

2.2.3.6.1 Output

For a sample input tuple (car, 2012, midwest, ohio, columbus, 4000), the above query
with cube and rollup operation will output

(car,2012,midwest,ohio,columbus,4000)
(car,2012,midwest,ohio,,4000)
(car,2012,midwest,,,4000)
(car,2012,,,,4000)
(car,,,,,4000)
(car,2012,midwest,ohio,columbus,4000)
(car,,midwest,ohio,columbus,4000)
(car,,midwest,ohio,,4000)
(car,,midwest,,,4000)
(car,,,,,4000)
(car,,midwest,ohio,columbus,4000)
(car,,midwest,ohio,,4000)
(car,,midwest,,,4000)
(car,,,,,4000)
2.2.3.6.2 Output Schema

```
grunt> describe cubed_and_rolled;
```

2.2.3.7 Handling null values in dimensions

Since null values are used to represent subtotals in cube and rollup operation, in order to differentiate the legitimate null values that already exist as dimension values, CUBE operator converts any null values in dimensions to "unknown" value before performing cube or rollup operation. For example, for CUBE(product, location) with a sample tuple (car, null) the output will be \{(car, unknown), (car, null), (null, unknown), (null, null)\}

2.3 MR-Cube for Holistic Measures in Apache Pig

The naïve method scales well for algebraic measures but for holistic measures it will result in very slow reducers or job getting failed due to out-of-memory exception. The reason is that there is going to be one reducer that gets all the tuples from input relation for computing the grand total. In order to overcome this reducer slowness/failure, the second phase of implementation focused on MR-Cube implementation [24] for distributed cubing on holistic measures. The following sections primarily focus on the
challenges in implementing MR-Cube algorithm and the techniques we used for overcoming those challenges.

2.3.1 Lack of Statistics

By default Apache Pig doesn’t store any statistics about the input data. The storage layer provides interfaces (LoadMetadata/StoreMetadata) for loading and storing the statistics along with the data. Pig also has integration with HCatalog [42], which is a metadata, table and storage management service for data created using Hadoop. In order to take advantage of statistics provided by HCatalog the data has to be stored and retrieved via specialized loader and storer (HCatstorer and HCatloader respectively). If statistics about the input data is not available through any of the above methods, MR-Cube algorithm will use an additional map-only job for gathering the statistics about the input data.

Following is the MR-Cube algorithm from [24]

MR-CUBE(Cube Lattice C, Dataset D, Measure M)

1. Dsample = SAMPLE(D)
2. RegionSizes R = ESTIMATE-MAPREDUCE(Dsample, C)
3. Ca = ANNOTATE(R, C) # value part. & batching
4. while (D)
5. do R ← R U MR-CUBE-MAPREDUCE(Ca, M, D)
6. D ← D # retry failed groups D’ from MR-Cube-Reduce
7. Ca ← INCR
Line 1 is a sampling job. This sampling job is required to estimate the regions in the cube lattice that are expected to be large. At the end of the sampling job large groups are determined and partition factor for those large groups are computed. In order to find the partition factor, total number of tuples in the dataset is required. Since the input dataset is distributed across multiple nodes we do not know the exact number of rows in the input dataset. To determine the total number of rows in the input data a new map-only job is added to the beginning of the MapReduce plan. This map-only job reads the entire dataset, materializes the tuples and serializes it back to Hadoop Distributed File System (HDFS). This serialization of tuple back to HDFS is facilitated by a custom intermediate storage layer (implements LoadMetadata/StoreMetadata interface), which along with the data also stores the total number of tuples read/serialized in a pig stats file. At the end of this map-only job all mappers will write statistics about the input split that it processed.

2.3.2 Low Overhead Sample Loader

The second map-reduce job is the actual sampling job. The sampling job is required to determine the partition factor for large groups in the cube lattice. Before starting the sampling job, all pig stats files stored as a result of previous map-only job are collectively read and the results are combined to get the overall combined statistics for the input data. This overall statistics will be used at the end of sampling job to determine partition factor for each regions in the lattice. With reference to MR-Cube paper [24] a sample size of 2M rows is sufficiently large for getting accurate estimate of up to 20 billion rows dataset. Since we often need only very less data for sampling when compared to the
entire dataset, it will be beneficial to have a random sample loader which does not read the entire dataset to keep the overall running time of sampling job minimal. The default random sample loader provided in Pig uses reservoir sampling technique to load \( n \) sample tuples. Since reservoir sampling reads the entire dataset to get a sample of \( n \) tuples, using the default random sample loader will become very expensive and will add too much overhead to the total job execution time. Hence we implemented a custom sample loader called CubeSampleLoader that wraps all the input splits from previous map-only job into a single logical split. This makes sure that only one mapper is launched for the sampling job. When this single mapper requests tuple from the CubeSampleLoader, the CubeSampleLoader randomly reads tuple from one of several (max 100) internal open splits for up to a maximum of the requested number of tuples. Thus CubeSampleLoader makes sure only the required number (2M) of tuples are randomly read without reading the entire dataset thus keeping the overhead of sampling job very low. After the samples are loaded, naïve cubing is performed over the sample dataset and large groups within each cube regions are determined. If the number of tuples in the large group exceeds a threshold \( (t) \) then that group is identified as reducer unfriendly group for which partition factor \( (p) \) will be determined. The threshold \( (t) \) is computed using the following equation

\[
t = 0.75rN
\]

where, \( r \) is maximum number of tuples a reducer can handle given by \( c / |D| \),
$c$ is the maximum number tuples a reducer can handle, $|D|$ is the overall data size, $N$ is the sample size. At the end of sampling job, partition factor ($p$) for each region in cube lattice is computed using the formula

$$p = \frac{s}{rN}$$

where, $s$ is the size of the group within a region, $N$ is the sample size, $r$ is maximum number of tuples a single reducer can process.

Each region in the cube lattice is then annotated with partition factor and the annotated lattice is persisted in a file in the HDFS.

Now that we are done with statistics gathering and sampling job, we can start materializing the actual cube job. The third map-reduce job is the actual cube computation job. Before the start of this job, the annotated lattice computed at the end of previous sampling job is shipped to each of mappers using Hadoop’s distributed cache.

Using this annotated lattice, each mappers appends the partition key modulus partition factor to the end of each tuple. This makes sure that the tuples belonging to large groups are partitioned approximately equally across the available reducers. The partition key is specific to the type of holistic measure being used. For example, if we would like to count the number of unique users in a particular region over a period of time, then the following pig latin script can be used

```pig
inp = LOAD '/pig/data/qlog' USING PigStorage('') AS (userid:int, region:chararray, state:chararray, city:chararray, month:int);
cubed = CUBE inp BY CUBE(region, month);
```
result = FOREACH cubed {
    dist = DISTINCT cube.userid;
    GENERATE FLATTEN(group), COUNT(dist) as unique_users;};

In the above script the measure is COUNT + DISTINCT is a holistic measure. The partition key for this holistic measure will be automatically identified as ‘userid’ since we are counting the number of unique users. If such partition key is determined then holistic measures will become algebraic for the identified attribute. [24] describes such an attribute as partially algebraic attribute and describes such a measure as partially algebraic measure.

After distributing the tuples from large groups to multiple reducers, the COUNT + DISTINCT measure is performed on each of the reducer. Since partitioning makes sure that disjoint set of tuples will available at each reducer the holistic measures can be safely used on the reducers. Since the results of large groups are spread across multiple reducers, a final follow up post-aggregation job is launched for computing the combined overall aggregated output from multiple reducers.

The following flowchart summarizes the sequence of jobs generated for the pig latin script mentioned above
Thus cube materialization can now be performed offline for algebraic and holistic measures using Apache Pig. The following section will cover online cube materialization using DICE, a distributed and interactive cube exploration system.
Chapter 3: Online Cube Materialization

The previous chapter discussed cube materialization in an offline context, which has the advantage of directly answering queries from the fully materialized cube. But offline cube materialization has a disadvantage of high storage cost and high processing time. The storage cost of fully materializing a data cube is exponential to the number of dimensions specified in cube query. Also analysis over a new measure requires recomputation of the entire cube. For example, if the cube is materialized for SUM measure and if the analyst wishes to change the measure to AVG then it is not possible without materializing the cube again. In order to alleviate these problems; cube materialization can be done in an online and interactive manner over a sample of data and get approximate results as soon as possible. This motivates us to build an online cube materialization system that overcomes the disadvantages of offline cube materialization system.

Online cube exploration is often expected to be interactive – queries need to be responded to within a small latency bound. Studies in the Human-Computer Interaction domain [23, 32] establish guidelines and demonstrate the functional and economic value of rapid response times, heavily motivating a sub-1000ms threshold for the database to respond to the user. For our system, we empirically observed that latencies of up to
1000ms were perceived as fluid, and with a 5000ms threshold on the time it takes the user to react to the query results. While such latencies are highly desirable to the end-user, they are often intractable over the scale of data being queried against. Thus, an often-used approach is to execute the query over a fixed subset of the data and provide an approximate answer while guaranteeing fluid exploration of the data cube. However, such an approach introduces its own problems – not all measures are useful over sampled data, and a query-specific sub-setting of the data is in itself a challenge.

We introduce DICE, a system that proposes a novel session-oriented approach to data cube exploration that caters to the challenges observed. It combines techniques in speculative execution of ad-hoc queries and data sampling to provide latencies that correspond to a fluid and interactive cube exploration experience.

Figure 3 shows the problem setting of DICE system. User Interface (UI) on analytics frontends have tunable sampling rates, and actions are translated to facet traversals (Section 3.3.1) over the data cube, representing ad-hoc queries. The master executes these queries over distributed slaves nodes (Section 3.4). As part of the DICE approach (Section 3.7), the master manages session state, query speculation and result aggregation, while the slaves perform query execution and caching. For each ad-hoc query, the master distributes the query to each slave, which may have some queries speculatively executed and cached. Results from each slave are then aggregated and error bounds are calculated, returned to the client, which then updates the UI.
In order to better articulate the challenges at hand, and the approaches used to address them, we now describe a motivating example.

### 3.1 Motivating Example

One typical use of interactive cube exploration is in the *management of cloud infrastructure*. For each setup, a handful of operations personnel manage tens of thousands of nodes, each with multiple virtual machines. Each instance produces a plethora of events, which are logged to track performance, detect failures and investigate systems issues. Each event item can be understood as a tuple with several fields, and each analytics task can be considered as a projection on a cube over the entire dataset. Event log data is copied over from all instances into a distributed data store, and is typically queried within fixed time ranges. Queries are ad-hoc, and due to the critical nature of the task, a system that allows for fast, interactive aggregations is highly desirable. An example query in our use case is:
SELECT rack, AVG(iops)
FROM events
WHERE datacenter = "EU" AND hour = 6
GROUP BY rack;

Such a query can be used to identify problematic I/O rates across racks, which could cause failures in a datacenter over time. We expect such queries to be either written by the operations personnel directly, or be generated automatically by applications that provide visualizations and an easy-to-use querying layer. An important insight is that such a process is not about aiding exploration such that user intervention is not required, but about helping the user analyze data faster by reducing the time it takes to interact with the data. Our driving use case possesses some unique characteristics that are not served by conventional database solutions. First, querying is ad-hoc and exploratory. Given the variety of possible questions to be answered, it is difficult to implement such a system over traditional reporting platforms, streaming queries, incrementally materialized views or query templates. Second, the data is distributed due to its size and nature of generation: events from each node in the datacenter are copied over to a set of nodes dedicated to this ad-hoc analysis to be used by one or few people. Another consequence of the size of the data is that it is impractical to construct a fully materialized cube to perform analysis. Fourth, user interaction, either through the application interface or through direct querying should not impede the user in performing their exploration task. Thus the interaction needs to be fluid, requiring the underlying queries to return quickly,
enforcing the latency bounds discussed above. Given the sampling requirements of the user, it is imperative that the results for the specified number of queries be returned at the earliest. Lastly, queries are never one-off, and almost always occur as part of a larger session of related queries. In light of this characterization, our problem thus becomes:

*Given a relation that is stored across multiple nodes, and a session of queries issued by the user, ensure that each query in the session is responded to at the earliest, at the user specified sampling rate.*

### 3.2 Common Approaches

Intuitively, the simplest approach to ensuring fast, interactive cube exploration is to materialize the entire data cube such that each query to the cube is simply a lookup from a main memory cache. While such a setup will perform within the latency bounds we are subject to, we are constrained by scale: a fully materialized cube can be several multiples of the original dataset (i.e. exponential to the number of dimensions), which typically is higher than available memory. Further, such a strategy does not work in the case of ad-hoc (e.g. computed) dimensions or if the user is inspecting a new measure. An alternative solution is to sample the dataset to create a working set that is small enough to run ad-hoc aggregate queries against. With a small enough dataset, we can meet interactive latency bounds. This approach, however, works for only fixed-size samples, and requires the entire small working set to reside on a single node to query against. Online aggregation approaches have also been considered, but require a significant overhaul of the entire query-processing infrastructure.
3.3 Data Model and Preliminaries

Having motivated the problem setting of a *distributed, interactive cube exploration* system, we now discuss preliminaries for each of the three contexts. We begin with cube exploration, where we define a *faceted exploration* model to facilitate complete yet efficient exploration of the data cube. Second, we discuss the fundamentals of execution of faceted queries in a distributed setting, where data is distributed across multiple nodes into *table shards*. Finally, given the constraints of interactivity, we introduce the notion of approximate querying over sampled data, provide a framework to execute faceted queries over multiple nodes, and draw from the concepts of stratified sampling and post-stratification to aggregate results and estimate error bounds. These three concepts define the context of our problem statement, and form the underpinnings of the *DICE* system, described in the following section.

3.3.1 Faceted Exploration of Data Cubes

In the context of cube exploration, the definitions of *cube, region, and group* are as per the original data cube paper [12] or Section 1.1. For exemplification, we continue our motivating example, using the following schema: Database table ‘events’ catalogs all the system events across the cluster and has three dimensions, two of which are hierarchical:

*location[zone:datacenter:rack], time[month:week:hour], iops*

Figure 4 shows the cube lattice showing valid regions corresponding to the dimensions listed above.
3.3.1.1 Challenges in Exploration:

As a user exploring a data cube, the number of possible parts of the cube to explore (i.e. cube groups) is very large, and thus, exploration can be unwieldy. To this end, we introduce the *faceted* model of cube exploration, which simplifies cube exploration into a set of facet traversals, as described below. As we will see in the following section, the faceted model drastically reduces the space of possible cube exploration and simplifies speculative query execution, which is essential to the *DICE* architecture.
We introduce the term *facet* as the basic state of exploration of a data cube, drawing from the use of category counts in the exploratory search paradigm of faceted search [35]. Empirically, most visualization such as map views and bar charts found in visual analytics tools can be constructed from aggregations along a single dimension. Facets are meant to be perused in an interactive fashion – a user is expected to fluidly explore the entire data cube by successively perusing multiple facets.

Intuitively, a user explores a cube by inspecting a facet of a particular region in the data cube – a histogram view of a subset of groups from one region, along a specific dimension. The user then explores the cube by traversing from that facet to another facet. This successive facet can be a **parent** facet in the case of a *rollup*, a **child** facet in the case of a *drilldown*, a **sibling** facet in the case of a *change of a single dimension value in the group* and a **pivot** facet in the case of a *change in the inspected dimension*. Thus, the user is effectively moving around the cube lattice to either a parent region, or a child
region or remaining in the same region using sibling and pivot traversals to look at the same data in a different way. Each session comprises of multiple traversals. The formal definitions are as follows.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f$</td>
<td>facet</td>
</tr>
<tr>
<td>$g$</td>
<td>group</td>
</tr>
<tr>
<td>$r(d_{i\ldots n})$</td>
<td>cube regions of dimensions $i \ldots n$</td>
</tr>
<tr>
<td>$d_i(g_a)$</td>
<td>$i^{th}$ dimension of group $g_a$</td>
</tr>
<tr>
<td>$d_b : v_b$</td>
<td>bound dimensions $d_b$ and corresponding measures $v_b$</td>
</tr>
<tr>
<td>$d_g$</td>
<td>grouping dimension</td>
</tr>
<tr>
<td>$(d_g)[d_{b1} : v_{b1}]$</td>
<td>facet consisting of grouping dimension $(d_g)$ and bound dimension $d_{b1}$ the corresponding value $v_{b1}$</td>
</tr>
</tbody>
</table>

Table 1: List of notations used in this section

**Facet:** For a region $r$ in cube $C$, a facet $f$ is a set of groups $g \in r(d_{i\ldots n})$ such that the group labels differ on exactly one dimension $d_i$, i.e.
∀g_a, g_b ∈ f, d_i(g_a) ≠ d_i(g_b) \land d_j(g_a) = d_j(g_b) \text{ where } i ≠ j \text{ and } d_i \text{ is the grouping dimension, and the remaining dimensions are the bound dimensions. In its SQL representation, a facet in a region contains a GROUP BY on the grouping dimension and a conjunction of WHERE clauses on the bound dimensions of that region. A facet can be referred to using the notation } f(d_g, d_b : v_b) \text{ where } d_g \cup d_b \text{ denotes the dimensions in the corresponding region, } d_g \text{ denotes the grouping dimension, } d_b : v_b \text{ denotes a vector representing the bound dimensions and their corresponding values. Thus, given the region } \{ \text{zone, week, iops} \}, \text{ the measure COUNT on the facet } (iops)[\text{zone: } z_1, \text{ week: } w_1] \text{ provides a histogram of I/O failure types at a zone during a specific week.}

**Facet Session:** A facet session } F \text{ is an ordered list of facets } f_{1,n} \text{ that a user visits to explore the data cube. The transition from one facet to another is known as a traversal.}

We now define four traversals, Parent, Child, Sibling and Pivot, each allowing us to move from one facet to another. We define them in terms of the destination facet, as follows.

**Parent Facet:** A parent facet is defined as any facet obtained by generalizing any of the bound dimensions.

Thus, a facet } f_p(d_{pg}, d_{pb} : v_{pb}) \text{ is a parent to the child facet } f_c(d_{cg}, d_{cb} : v_{cb}) \text{ if } d_{pg} = d_{cg} \text{ and } d_{pb} : v_{pb} \text{ represents a parent group of } d_{cb} : v_{cb} \text{ in the cube lattice. The facet } (iops)[\text{zone: } z_1, \text{ month: } m_1] \text{ generalizes the dimension time from the prior example (month : m_1 was the implicit higher hierarchical dimension for the bounded dimension
week : w₁), and is therefore its parent facet.

**Child Facet:** Similarly, a child facet is defined as any facet obtained by specializing any of the bound dimensions. The facet \((iops)[zone:z₁, hour:h₁]\) specializes the dimension \(time\) from the facet example, and is therefore its child facet.

**Sibling Facet:** A sibling facet is defined as any facet obtained by changing the value for exactly one of the bound dimensions.

Thus, a facet \(f_x(d_{xg}, \overrightarrow{d_{xb}}: \overrightarrow{v_{xb}})\) is a sibling to the facet \(f_y(d_{yg}, \overrightarrow{d_{yb}}: \overrightarrow{v_{yb}})\) if \(d_{xg} = d_{yg}\), \(\overrightarrow{d_{xb}} = \overrightarrow{d_{yb}}\) and \(\overrightarrow{v_{xb}}\) and \(\overrightarrow{v_{yb}}\) differ by exactly one value. The facet \((iops)[zone:z₁, week:w₂]\) changes the value of \(week\) from the facet example, and is therefore its sibling facet.

**Pivot Facet:** A pivot facet is defined as any facet obtained by switching the grouping dimension with a bound dimension.

Thus, a facet \(f_x(d_{xg}, \overrightarrow{d_{xb}}: \overrightarrow{v_{xb}})\) can be pivoted to the facet \(f_y(d_{yg}, \overrightarrow{d_{yb}}: \overrightarrow{v_{yb}})\) if \(d_{yg} \in \overrightarrow{d_{xb}}\), \(d_{xg} \in \overrightarrow{d_{yb}}\) and \(\overrightarrow{v_{xb}}\) and \(\overrightarrow{v_{yb}}\) have all but one bound dimension (and value) in common. The facet \((zone)[week:w₁, iops:i₁]\) pivots on \(iops i₁\) from the facet example, and is therefore its pivot facet.

**3.3.1.2 Explorability of the cube:**

It is critical that the user be able to *fully explore* the data cube, i.e. all cube groups can be explored using facets, and it is possible to reach any facet from any other facet.
First, for a group \( g = d : v \), there can be \( |d| \) facets, \( f(d_g, d_b : v_b) : d_g \in d \land d_b = d - d_g \). Second, any two facets in a region can be reached from another by a series of sibling and pivot traversals: sibling traversals to change bound values, and pivot traversals to switch between bound and grouped dimensions. Parent and child traversals allow us to reach the corresponding parent and child regions in the cube lattice. Thus, the four traversals enable full exploration of the cube lattice.

3.4 Distributed Execution

3.4.1 Table Shards

We use sharded tables to achieve distributed and sampled execution of queries. A sharded table is the atomic unit of data in our system, and contains a subset of the rows of a SQL table and the concatenation of all shards across nodes is equivalent to the entire dataset. Each single node may contain multiple shards. Updates are atomic to each shard, and each session makes the assumption that the list of shards and the shards themselves do not change.

3.5 Querying over Sampled Data

As mentioned before, the interactive nature of our use case necessitates the approximation of results by executing queries over a sample of the data. We need to be able to retrieve results by accessing a fraction of the total data and provide results with error bounds. In this section, we draw from the concepts of stratified sampling and post-
stratification and present a methodology for doing so.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_h^2)</td>
<td>variance of group (h)</td>
</tr>
<tr>
<td>(n_{hi})</td>
<td>number of tuples belonging to group (h) from the (i^{th}) query</td>
</tr>
<tr>
<td>(m_{hi})</td>
<td>mean of the group (h) from the (i^{th}) query</td>
</tr>
<tr>
<td>(m_h)</td>
<td>mean of the group (h) from all queries</td>
</tr>
<tr>
<td>(\nu_{hi})</td>
<td>variance of the group (h) from the (i^{th}) query</td>
</tr>
<tr>
<td>(\nu[\bar{x}])</td>
<td>variance of the estimator for the measure (\text{SUM})</td>
</tr>
<tr>
<td>(\nu[\bar{y}])</td>
<td>variance of the estimator for the measure (\text{AVG})</td>
</tr>
<tr>
<td>(\nu[\bar{p}])</td>
<td>variance of the estimator for the measure (\text{COUNT})</td>
</tr>
<tr>
<td>(H)</td>
<td>Total number of groups in the union of all the queries</td>
</tr>
<tr>
<td>(N_h)</td>
<td>Total number of tuples in the combined group (h)</td>
</tr>
<tr>
<td>(n_h)</td>
<td>Number of tuples in the sample in the combined group (h)</td>
</tr>
<tr>
<td>(N)</td>
<td>Total number of tuples in the dataset</td>
</tr>
<tr>
<td>(p)</td>
<td>Proportion of tuples selected by the where clause</td>
</tr>
</tbody>
</table>

Table 2: List of notations used in this subsection

In order to deliver results at higher sampling rates, \(DICE\) runs the same query on multiple shards on multiple nodes. This results in the same cube group of the facet query being
possibly obtained from the multiple table shards. Hence, the statistics for the same group from these multiple queries need to be combined together. While combining the AVG, SUM and COUNT are straightforward; the variances can be combined as shown in Appendix A as:

\[
s_h^2 = \frac{1}{n_h - 1} \left( \sum_{i=1}^{numQ} n_{hi} (m_{hi} - m_h)^2 + \sum_i (n_{hi} - 1) v_{hi} \right)
\] (1)

where \( numQ \) is the number of queries that a facet query needs to be replicated to, to achieve the user specified sampling rate.

Continuing our motivating example, the faceted representation of the query is (rack)[hour:6,datacenter:EU] with the measure and measure dimension being AVG(iops). We append the COUNT and VARIANCE measures to the queries since we need them as described in Equation 1. Let this query be run on a single shard on a couple of nodes and resulting into a sampling rate of 10%, returning us groups and the corresponding measures from the two queries respectively as:

\[
\{[\text{rack:1,hour:6,datacenter:EU,AVG:10,COUNT:5,VAR:4}],
\text{rack:2,hour:6,datacenter:EU,AVG:12,COUNT:6,VAR:2]}\} \quad \& \quad \{[\text{rack:1,hour:6,datacenter:EU,AVG:5,COUNT:8,VAR:1}],
\text{rack:2,hour:6,datacenter:EU,AVG:6,COUNT:7,VAR:2]}\}.
\]
Plugging in the values from above into Equation 1, we would get the variance for the combined group \([\text{rack:1,hour:6,datacenter:EU}]\) as \(s^2_1 = 8.32\) and for \([\text{rack:2,hour:6,datacenter:EU}]\) as \(s^2 = 11.52\).

The standard deviation can be used as an error estimate for the entire query. Once we know the variance of each of the combined groups, we can get an error estimate for the combination of all of these groups i.e. the combined result set. We consider three measures SUM, AVG and COUNT. The variance of the estimator for the measure SUM can be given as:

\[
\mathcal{V}[\hat{\varepsilon}] = \sum_{h=1}^{H} N_h^2 (1 - \frac{n_h}{N_h}) \frac{s_h^2}{n_h} \quad (2)
\]

Similarly, variance of the estimator for the measure AVG:

\[
\mathcal{V}[\hat{\gamma}] = \sum_{h=1}^{H} \frac{N_h^2}{N^2} (1 - \frac{n_h}{N_h}) \frac{s_h^2}{n_h} \quad (3)
\]

Continuing with our example, we estimate \(\frac{N_h}{N}\) by \(\frac{n_h}{n}\) and \(\frac{n_h}{N_h}\) by the sampling rate since we cannot know \(N_h\) without sampling the entire data.

Again plugging in the values we get, \(\hat{\gamma} = 6.92 \times \frac{13}{26} + 8.77 \times \frac{13}{26} = 7.85\) and
\[ \hat{\mathcal{V}}[\hat{\epsilon}] = \left( \frac{13}{26} \right)^2 \times (1 - 0.1) \times \left( \frac{8.32}{13} + \frac{11.52}{13} \right) = 0.35 \]

For the measure COUNT, we can use the proportion estimator

since the where clause acts as the indicator function and thus the variance of the estimator for COUNT can be given as:

\[ \hat{\mathcal{V}}[\hat{\rho}] = \frac{(1 - \frac{n}{N}) \hat{\rho}(1 - \hat{\rho})}{n - 1} \quad (4) \]

The error will be given as a ratio of half of the width of the Confidence Interval to the Estimate:

\[ \frac{\text{ConfidenceInterval}}{2 \times \text{Estimate}} = \frac{z_{\alpha/2} \sqrt{\mathcal{V}(\hat{\theta})}}{\hat{\theta}} \quad (5) \]

For a confidence of 95%, we can get the standard error for the query as \(1.96 \times \frac{\sqrt{0.35}}{7.85} = 0.15\). Thus, the accuracy of the query will be 85%, which is returned to the user. The user can then use this information to explore other facets or change the sampling rate should they be interested in a more fine-grained version of the result.

### 3.6 Sampling and Result Quality

Given the preliminaries and definitions in the previous section, in the naive case, the problem of ad-hoc cube exploration using the facet exploration model is simply that of
successively executing each query as it comes along, at a certain sampling rate. We formulate our problem of retrieving results quickly for large datasets at high sampling rates as the following:

*For a facet session \( F \), where each ad-hoc facet query \( f_i \) is expected to execute at a certain sampling rate, and the time between the termination of one facet query and the start of the next ad-hoc facet query (i.e., the time taken to view the results of the prior query) is \( \tau \), return \( f_i \) as quickly as possible to the end-user, preferably within the interactive threshold \( \tau_i \).*

Since an exact value for the sampling standard error reduction by sampling an additional rate cannot be known before we actually sample the data, we need a metric to approximate it. We use a heuristic based on maximum likelihood estimation’s consistency property, which is \( ||\theta^* - \theta|| = O \left( \frac{1}{\sqrt{n}} \right) \) where \( \theta^* \) is the current estimate, \( \theta \) is the true value and \( n \) is the current sampling rate, which informs us that the difference between our estimate and the true value will be inversely proportional to the square root of the current sampling rate, for increasing sampling rates.

Therefore, we can estimate the future gain in accuracy based on the sampling rate. The estimated gain in the accuracy can be given by the difference of the estimated improved accuracy obtained by increasing the sampling rate, \( R_{inc} \), with the one at the current sampling rate, \( R_{curr} \).
\[
\Delta EstimatedAccuracy(R_{inc}, R_{curr}) = c \times \left( \frac{1}{R_{curr}} - \frac{1}{R_{inc}} \right) \tag{6}
\]

where \(c\) is the constant from the proportionality heuristic.

With more time permissible, we issue the same query on multiple tables on multiple nodes giving us progressively a smaller standard error for the estimators. Our goal then during speculative execution of the queries is to increase the likelihood that the next user query would be cached at a higher sampling rate allowing us to retrieve the results at the desired sampling rate at the earliest. We further reformulate this problem to fit the DICE framework in the following section.

### 3.7 The DICE System

#### 3.7.1 Speculating Queries in a Session

A crucial insight to ad-hoc querying is that queries occur in sessions. Thus, it is prudent to think of improving query performance holistically at the session-level. A session comprises several ad-hoc queries, each of which requires low-latency responses. The user inspects the result for each query for a small amount of time, after which a successive query is issued. We consider this as a hidden opportunity – the database is simply waiting on the user to issue the next query. In light of this, our solution is to \textit{utilize this waiting time to speculate, execute and cache the most likely follow-up queries at the highest quality possible}. While the concept of speculative execution is an intuitive one, there are
several challenges of implementing it over a distributed, approximate querying environment – especially in the context of data cube exploration. The challenges comprise a host of interdependent problems: What are the most likely follow-up queries? What is the strategy to employ to execute and cache likely queries? In a sampling approach, what is the highest sampling rate to run a speculative query at, given interactive constraints? Finally, is there a singular framework to combine these problems into a cohesive, unified system?

Given these challenges, we present the DICE system that solves the problem by using three complementary strategies. First, it performs speculative query execution, by caching results of likely follow-up queries, allowing for reduced latencies for ad-hoc query sessions. The enumeration of the likely follow-up queries is made possible by the faceted model of data cube exploration described in Section 3.3.1.1. Second, DICE employs a novel architecture of query execution over a distributed database, executing queries piecemeal over individual table shards and then assembling them in a post-processing step. This novel architecture in turn allows for bounded-time execution of queries ensuring interactive latencies. Third, it employs a cost-based model for the prioritized execution of speculative queries such that likely queries are executed at higher sampling rates.
3.7.2 System Architecture

In this section, we will outline the approaches that we used for building distributed query execution engine. Execution engine is the heart our DICE system and we built it while keeping the following core properties in mind:

- **Low latency** – In order to keep the overall execution time under sub-second, each component of the system should have minimal overhead since it directly affects the overall execution time.

- **High throughput** – The execution engine should process as many queries as possible within the bounded time.

- **Scalable** – The entire system should scale horizontally as more nodes are added to the system.

- **Reliable** – Since query execution is distributed in nature, reliability of query result returned from each node is of high importance, as it will impact the error bounds of the query.

- **Bounded time execution** – The execution engine should stop processing queued requests after the expiration of time bounds. This is essential for speculative queries since we can only execute subset of speculated queries within 5000ms view latency ($\tau_y$).

Figure 6 shows our first approach towards implementing distributed execution engine. In this approach, there is a single master node, which maintains a thread pool of JDBC connections to remote slave nodes. The entire dataset is horizontally partitioned and
distributed across all the slave nodes. When a query is submitted to master node, the query federator replicates the query based on the specified sampling rate and the available number of slave nodes and creates a list of queries that needs to be executed on each slave node. The threads in the thread pool then execute the list of queries on each of the slave nodes using direct JDBC connections. The resultsets are then aggregated and sent back to the client. The query federator internally maintains a timer to guarantee bounded time execution.

Approach 1 though achieves low-latency, high throughput, reliability and bounded time execution properties, it suffers from scalability because of the threaded model. Since the number of threads is linearly proportional to the number of slave nodes, when the number of slave nodes increases the memory overhead of thread-based model will increase significantly. To get a better overview, each thread in a 64-bit JVM takes approximately 1MB of memory. Master node maintains 4 threads\(^4\) per node in the thread pool. So for 50 nodes the total memory used by thread pool will be approximately 200MB, which is expensive.

\(^4\) For optimal performance of PostgreSQL server, the maximum number of connections should be equal to the number of CPU cores. All slave nodes have Quad core CPUs and hence 4 JDBC connections per node are optimal.
Another disadvantage with Approach 1 is that memory of slave nodes are never used, which can be used for caching the resultsets. This motivated us to implement master-slave hierarchy for query execution as shown in Figure 7. With this master-slave approach we are able to make use of LRU cache on each slave nodes for caching resultsets. Also the high memory overhead of thread pool is distributed across the slave nodes. The master node maintains only one thread per slave node and each slave node maintains 4 threads, which are JDBC connections to database. This approach scales relatively better than Approach 1 but we were not able to get fine-grained control over time bound execution. Also this approach suffers from blocking code at the master end. Each thread blocks until query execution is completed and results are returned back to master.
In order to overcome the issues with Approach 1 and 2, we rewrote the execution engine using event-based Actor Model. Actors are very lightweight concurrent entities that respond to messages. Each actor in an actor system occupies only 300 bytes of memory as opposed to ~1MB memory for threads. Also actor creation is extremely fast when compared to thread creation. Each actor has a mailbox and they process and respond to messages received in their mailbox. We used Akka library, which is an implementation of Actor Model for JVM. Akka is an event driven middleware for building highly concurrent, distributed and fault tolerant applications. With this, we built a complete

Figure 7: Threaded Master-Slave DICE Architecture (Approach 2)
asynchronous event-based distributed execution engine that processes messages (containing queries). As can be seen from Figure 8 master node exchanges messages between slave nodes and slave nodes respond back with message containing result of the processed query.

In this new actor model based architecture, we employ a hierarchical master-slave approach, such that all queries are issued to the master, and responded to by the master. Each slave manages multiple table shards. Each shard is atomic and read-only, and is implemented as a table in a commodity relational database. The catalog of shards across all slave nodes is maintained at the master. For a single exploration session, the catalog is used to ensure that the list of shards addressed is constant. The slaves maintain an in-memory LRU cache for the results. In a fast-changing database, table shards can be atomically added and deleted from the slaves, and the master’s catalog can be updated, allowing for querying over rapidly changing data. After scheduling the view queries to each slave node, the master node aggregates the results and sends back final aggregated result back to the client. After sending the result of view query, master node schedules speculative queries to slave node routers. Routers are actors, which manages 4 worker actors. Messages received by these routers are scheduled using smallest-mailbox assignment to all workers. Smallest mailbox based task assignment enables proper load balancing of speculative queries across workers. In order to guarantee bounded time execution of speculative queries, we implemented a custom mailbox for routers that will discard the messages in the mailbox after 5000ms timeout ($\tau_V$). With this new architecture, we were able to achieve all the desired properties as outlined in the
Following section will cover the query flow and sequence of operations from frontend application to backend execution engine.

### 3.7.3 Query Flow

The high-level query flow of DICE is as follows: each ad-hoc query is rewritten and federated to the slave nodes, where it is executed. The results are returned, aggregated
and presented to the user, along with the accuracy of the query. Upon success, a set of speculative queries is executed within a bounded time, with the goal of caching as many of the future queries as possible. When the successive ad-hoc query is issued, it is again rewritten and federated, with the hope that a majority of the queries are cached at the slaves, thus reducing the latency of the overall ad-hoc query.

At startup, the master makes sure that all the slaves are running and ready to accept queries. On receiving an ad-hoc query, the query is rewritten into multiple queries, one per required table shard and passed to each slave. Since data is horizontally distributed across all slave nodes, the query itself is identical, with the exception of id of the table shard addressed. On completion of an ad-hoc query (or if the results of the query were already in the cache), each slave returns the results back to the master, where the results are aggregated, and along with the error calculation (due to sampling, as discussed in the previous section), and presented to the user.

Upon completion of the ad-hoc query, the master immediately schedules a list of speculative queries that can be issued by the user. While the space of possible queries is unbounded, we restrict our speculations using faceted exploration framework; thus allowing the list of possible queries to be enumerable. Speculated queries are then ranked (as discussed in the following subsection), and the ranked list is sent to each slave, which employs bounded time execution – each slave issues a predefined number of concurrent queries to its database and populate results in the cache (speculative query results are not sent to the master). Once the speculation latency threshold has passed the view latency
threshold ($\tau_v$), the slave kills all currently running queries and clears the query queue.

When the next ad-hoc query arrives, it is again rewritten and federated to the slaves. If the exact query or a unified query (refer section 3.8) is cached, the result of the ad-hoc query is materialized from the cached result. If it is not cached it is then executed on the database. For example, if a query is to be run over 20 table shards and 18 of them are cached, the remaining 2 are executed on the database, and all the results are returned. The caching of speculated queries drastically impacts ad-hoc query latency and allows for a fluid, interactive data cube exploration experience.

### 3.7.4 Prioritizing Speculative Queries

As is clear from the query flow, each ad-hoc query can yield significantly large number of speculative queries. Given the bounded time available for execution, it is typically not possible to evaluate all speculative queries. Thus, it is important for us to select a sub-set of speculative queries that maximizes the likelihood of queries on table shards being returned from the cache. This can in turn be done by maximizing the overall gain in accuracy, as discussed in Section 3.6. The selection of the maximal subset can be modeled as an integer-programming problem as follows:

\[
\text{Maximize: } \sum_{q \in Q} \text{Prob}(q) \cdot \text{AccuracyGain}(q) \cdot x_q \\
\text{Subject To: } \sum_{q \in Q} \text{Time}(q) \cdot x_q \leq \text{totalSpecTime} \\
\text{Where: } x_q \in \{0,1\}
\]
Here, \( \text{Prob}(q) \) gives the probability of a query, which is obtained from the data distribution with the intent of prioritizing populous cubes groups, \( Q \) is the set of all speculative queries at all sampling rates, \( \text{AccuracyGain}(q) \) is our estimate of the gain in accuracy which depends on the sampling rate of \( q \), \( \text{Time}(q) \) is the estimated running time of the query and \( \text{totalSpecTime} \) is the expected total speculative time (allotted to speculative query execution). Due to the size of the table shards, we expect the majority of the query execution cost to be typically due to an in-memory table scan over identically sized data. This lets us assume unit cost for all queries. Thus, it is clear that choosing the query that yields the maximum of the product of the probability of a query and the estimated accuracy gain for the corresponding sampling rate is the best decision. Thus, the solution to the problem of choosing of the \textit{best} queries that yield the highest overall accuracy gain turns into a greedy selection problem, the algorithm to which we provide in the following section. Should more detailed query cost estimates be available, it is trivial to incorporate them into our strategy by using the integer linear programming methodology instead of the greedy selection.

\textit{Greedy Approach:}

The greedy cost-based approach prioritizes the execution of \textit{the most likely queries that provide the highest overall accuracy gains}. We represent the \textit{score} of a query as

\[ \text{Prob}(q) \cdot \text{AccuracyGain}(q) \]

In the case of multi-query optimizations such as \textit{unification} (described in Section 3.6),
where multiple queries are grouped together into a unified query $Q = q_{1..n}$, the score can be represented as $\Sigma_{q \in Q} Prob(q) \cdot AccuracyGain(q)$. Queries are run greedily on the worker nodes in descending order of the score. Since worker nodes are capable of bounded-time execution and each query runs in a time lesser than the view latency threshold due to the small size of the table shard, this approach proves to be a viable strategy and successfully provides for sub-second latencies, as observed in Section 3.9.

One can also notice that in the case of sibling traversals, a user is more likely to choose the new changed bound dimension value to be close to the current value. We use a heuristic that the distribution can be given as $P(newVal) = O\left(\frac{1}{|newVal-oldVal|^2}\right)$. The distribution can be modeled using columns of the data from the prevalent task. Let the set of speculative sibling queries and their probabilities be $SQ = \{SQ_{1..n}\}$ and $P = \{P_{1..n}\}$ respectively. We redistribute the sum of these probabilities between $SQ$ as $P(x) = \frac{(x-oldVal)^2}{c}$ where $c$ is the normalization constant.

### 3.7.5 The DICE Algorithm

We are now able to illustrate formally the DICE algorithm as given in Algorithm 2 ($DICE$ Exploration Strategy). We first describe the overall model of the system as given below.
Algorithm 1: Core Exploration Loop

Algorithm 1 (Core Exploration Loop) describes the overall DICE cube exploration system. A user first selects a query, which is then executed (Lines 7-8). The system then enumerates all the different possible next queries based on the cube exploration model described earlier and then chooses a subset of them (Lines 9-10). It then distributes the workload across all the available nodes (Lines 11-13).

Next, in Algorithm 2 (DICE Execution Strategy) we formally describe how DICE chooses a subset of speculative queries to run. Algorithm 2 (DICE Execution Strategy) starts by first finding out the normalized probabilities $P_{spec}$ given a set of speculative queries (Line 1), and reweighting probabilities of the sibling queries as described in Section 3.4 (Line 2). Next, it generates the vector for increasing sampling rates (Line 3) and then performs unification over all the speculative queries (Line 4). Finally, it ranks
the unified queries at different sampling rates by the product of their probabilities and their corresponding sampling rate (Lines 5-11) and returns the top NT queries (Line 14).

\textbf{DICE-PLAN-DETERMINE}(Q_{\text{Spec}}, \tau_V, CF, NT)
\begin{enumerate}
\item $P_{\text{Spec}} \leftarrow \text{GET-SPEC-PROBABILITIES}(Q_{\text{Spec}})$
\item $PS_{\text{Spec}} \leftarrow \text{SIBLING-ADJUSTMENT}(P_{\text{Spec}}, CF)$
\item $\text{Accuracy.Gains} \leftarrow \text{GENERATE-ACCURACY-GAINS}()$
\item $\text{Unified.Questions} \leftarrow \text{QUERY-UNIFICATION}(Q_{\text{Spec}})$
\item $P_{\text{Unified.Questions}} \leftarrow \{\}$
\item \textbf{for each} $UQ$ \textbf{in} $\text{Unified.Questions}$
\item \textbf{do}
\item \hspace{1cm} $P_{UQ} \leftarrow \sum_{Q \in UQ} PS_{\text{Spec}}(Q)$
\item \hspace{1cm} $\text{Unified.Accuracies} = \text{Unified.Questions} \times \text{Accuracy.Gains}$
\item \hspace{1cm} $\text{DESC-SORT}((\text{Unified.Accuracies}))$
\item \hspace{1cm} \textbf{return} $\text{TOP}(\text{Unified.Accuracies}, NT)$
\end{enumerate}

Algorithm 2: DICE Execution Strategy

Algorithm 3 (Facet Selection) describes how we have modeled the user behavior. First, we choose the next traversal type given the current facet (Line 2) and then choose the varying dimension (Line 3). Next, we create the parent facet if the traversal type was the parent one (Lines 4-5). In other cases, we first create the where predicate (Line 7) and then the new facet (Line 8).
TRAVERSER (User \( u \), Facet \( CF \))
1    \(/ / T D \) is the Traversal Dimension
2    traversalType \( \leftarrow \) NEXTTRAVERSAL (CF)
3    \( T D \) \( \leftarrow \) CHOOSEDIMENSION (\( CF \), traversalType)
4    if traversalType = Parent
5       then nextFacet \( \leftarrow \) MAKENEXTACET (\( CF \), \( T D \))
6    else
7       pred \( \leftarrow \) MAKEPREDICATE (\( CF \))
8       nextFacet \( \leftarrow \) MAKENEXTACET (\( CF \), pred)
9    return nextFacet

Algorithm 3: Facet Selection

3.8 Optimization: Query Unification

The list of possible speculative queries, \( f_i \in \tilde{F} \) is derived from the number of possible
traversals for a facet \( f (d_g, d_b; v_b) \). For each traversal type, these are:

\[
\text{NumParent} = |\overrightarrow{d_b}|
\]

\[
\text{NumChild} = \sum_{\text{dim} \in \text{Dimensions} - \{d_b, d_g\}} \text{Cardinality} (\text{dim})
\]

\[
\text{NumSibling} = \sum_{\text{dim} \in \overrightarrow{d_b}} \text{Cardinality} (\text{dim}) - |\overrightarrow{d_b}|
\]

\[
\text{NumPivot} = \text{Cardinality} (d_g) \times |\overrightarrow{d_b}|
\]

Consequently, one can infer that the total number of the speculative queries, \( \text{SpecTotal} \),
could be greater than the sum of the cardinalities of non-hierarchical dimensions. Hence,
it is not feasible to run all the speculative queries for most real-world datasets at high
sampling rates within interactive time bounds, since the total number of speculative queries would be equal to the product of the number of table shards and SpecTotal.

However, we can observe that the generation of speculative queries leads to several queries that differ only by the value that a single bound dimension takes. Unifying multiple such queries into a lesser number of queries becomes essential since **concurrently running all of them will congest the system**. We have used two techniques of minimizing queries by unifying them. The first technique is unify WHERE clauses on a column into a GROUP BY on the column, and the second is to split a dimension’s domain into ranges, unifying queries across each range. Once executed, the results of these unified queries will be post-processed to extract results for the original query, if and when needed.

### 3.8.1 Group-by Based Unification:

Multiple queries can be unified into a single query by replacing the bound dimension that takes multiple values by a GROUP BY on the same dimension when the cardinality of a dimension is moderately high (i.e. above a set threshold). This unification leads to the following speculative queries for a facet $f(d_g, \overrightarrow{d_b}, v_b)$ in a cube of dimensions $\overrightarrow{d}$:

- **Parent Set** = \[ \forall d_i; d_i \in \overrightarrow{d_b} : f(d_g, \overrightarrow{d_b}, v_b - d_i: v_i) \]

- **Sibling Set** = \[ \forall d_i; d_i \in \overrightarrow{d_b} : f(d_g, d_i, \overrightarrow{d_b}, v_b - d_i: v_i) \]

- **Pivot Set** = \[ \forall d_i; d_i \in \overrightarrow{d_b} : f(d_g, d_i, \overrightarrow{d_b}, v_b - d_i: v_i) \]
\[
Child - Set = \{ \forall d_i; d_i \in \tilde{d} - \tilde{d}_b : f(d_g, d_i, \overline{d_b : v_b}) \}
\]

The sibling and pivot queries thus generated are identical. One can also notice that a parent query \( f(d_g, \overline{d_b : v_b} - d_i : v_i) \) can be completely answered by the corresponding sibling/pivot query \( f(d_g, d_i, \overline{d_b : v_b} - d_i : v_i) \) where \( d_i : v_i \in \overline{d_b : v_b} \). Thus, group-by based unification leads to an enormous reduction in possible queries needed to be run. However, it is clear that the results for the next query would need to be retrieved from the new unified query’s result set, a superset of the result set for the actual query, and this post-processing may be expensive. Typically, group-by unification is useful specifically for moderately high (preset thresh-old on empirical observations) cardinality dimensions. There is, clearly, a tradeoff between running a large number of non-unified queries, and a single unified query with a large result set.

### 3.8.2 Range Based Unification:

At very high cardinalities, there is a tradeoff between the time spent in executing speculative queries on the database and materializing/subsetting the result from cache. A useful solution is to unify the queries into ranges. Thus, we convert multiple speculative queries \( f(d_g, \overline{d_b : v_{1,n}}) \) into fewer range based speculative queries.

\[
f(d_g, \overline{d_b : v_{1,n1}}), \ f(d_g, \overline{d_b : v_{n1+1,n2}}),.. f(d_g, \overline{d_b : v_{n_k+1,n_k}}). \]

The choice of range based unification and group based unification is a tunable parameter on the column cardinality and is specific to the data at hand.
An interesting observation in the case of range queries is that even with careful tuning of the ranges, the cardinality of the data germane to each range-unified query is large enough to motivate the use of an index on range-unified columns. While this introduces variability into our cost model, the lack of a good determiner for the cost of a range-unified query compels us to invoke Occam’s Razor and use a unit cost in this case.

3.8.3 Optimality of DICE

As described in Section 3.7.4 due to approximately uniform cost of running a query, the linear integer programming problem of maximizing the overall accuracy under the constraint of maximum allocated time simply turns into a greedy algorithm of choosing the new query at an additional unit-sampling rate. And as also mentioned earlier in Section 2.4, since we would not know what the accuracy gain would be without actually running the query, approximating the gain using the estimated accuracy gain based on the MLE Consistency property is a sound assumption to make. Thus the DICE algorithm of choosing a new query with the highest product of probability from the workload and the estimated accuracy gain at the newer sampling rate will indeed be the optimal strategy.

To validate our assumption of the estimated accuracy gain being inversely proportional to the current sampling rate empirically, we ran a workload of queries for different measures over datasets with normal, zipfian, multinomial distributions and noticed that the sum of squared errors was the least for $\frac{1}{\sqrt{n}}$ amongst others like $\frac{1}{n}$ and $\frac{1}{n^2}$. 

60
3.9 Experiments and Evaluations

3.9.1 Experimental Setup

DICE is implemented in Java running on Sun Java 6 VMs and uses PostgreSQL 9.1 as the database for each slave node. By default, we discard the first run of each experiment and report the average of the following three runs. We perform an exhaustive analysis of the DICE system over a variety of cluster configurations, datasets, workloads and algorithms for our metrics, as described below.

3.9.1.1 Cluster Configurations

CLUSTERSMALL is a private cluster built on commodity hardware with only DICE running during the experiments. The master node has 1 Quad Core 3.30GHz Intel i5 CPU, 16GB DDR3 RAM @1333MHz & 256GB SATA HDD and the 15 slave nodes each possess 1 Quad Core 2.13GHz Intel Xeon CPU, 4GB DDR2 RAM @667MHz & 720GB SATA HDD. Nodes are connected over a Gigabit Ethernet switch. Each slave contains 4 workers. CLUSTERCLOUD is an Amazon EC2 configuration of 1 master and 50 slaves of the c1.xlarge type, each with 7GB Memory and 8 Virtual cores, powering 8 workers per slave node. All nodes for both configurations run Ubuntu Linux 12.04 LTS.

3.9.1.2 Datasets

Our generated dataset conforms to the example schema provided in Section 1 (Fig 2), and comprises 1 billion rows sharded uniformly across all nodes with a default table shard
The size of 1M rows. The distribution and cardinality for the dimensions are: location[uniform]:[zone{10}:datacenter{100}:rack{1000}], time[gaussian]:[month{12}:week{52}:hour{24}] and iops[zipfian]:{10000}. Each table shard is 102MB on disk, with a data size of 81MB and index size of 21MB, yielding in a total of 1000 table shards spanning 100GB. Unless otherwise specified, we run experiments at 20% sampling rate i.e., 200 million rows are actually processed\(^5\)

### 3.9.1.3 Workloads

We use a synthetic workload derived from queries logged from manual usage of a popular BI tool. The workload depicts a user query session of 10 ad-hoc facet traversals, with the measure function AVG. The interaction latency threshold \(\tau_i\) is set to 1000ms, and the viewing latency threshold \(\tau_v\) is set to 5000ms.

### 3.9.2 Algorithms

We compare five different algorithms: ALGONOSPEC stands for “No Speculation” and represents the baseline use case, i.e. ad-hoc distributed querying without any speculation, similar in design to modern distributed query execution engines. ALGORANDOM represents distributed querying using query speculation, but the queries chosen to be speculated are selected randomly from the set of possible facet traversals.

\(^5\) It should be noted that due to the variability of schema, row / columnar storage layouts and hardware performance, our focus is on the number of rows processed, and not the disk representation.
ALGOUNIFORM selects speculative queries uniformly from each type of facet traversal.

ALGODICE selects speculative queries based on the greedy cost-based heuristic, implements the unification heuristic and also predicts the next traversal using value locality. Finally, ALGOPERFECT “improves” upon DICE by allowing for a perfect prediction of the subsequent ad-hoc query – this represents the (hypothetical) best-case performance of our speculation strategy, and is included to demonstrate the overall potential of speculative caching.

3.9.3 Metrics

AVERAGE LATENCY is measured in milliseconds as the average latency of each query in a query session. We also depict ±1 standard deviation of latency using error bars in most of our results. AVERAGE ACCURACY is measured as the absolute %age deviation of the sampled results from the results over the entire dataset.
3.9.4 Results

3.9.4.1 Impact of Data Size

Figure 9: Varying Size of Dataset: CLUSTER_SMALL

Figure 10: Varying Size of Dataset: CLUSTER_CLOUD
We observe, in Figure 9 and Figure 10, the impact of data size on the latency observed by each algorithm by varying the target sample size for the ad-hoc queries in our workload. ALGONOSPEC scales approximately linearly, and fails to stay within the sub-second threshold for 200M rows. Despite issuing speculative queries, ALGORANDOM and ALGOUNIFORM perform just as poorly as ALGONOSPEC, validating the need for a principled approach to speculative querying that DICE provides. ALGODICE succeeds to stay within the sub-second threshold, and scales quite well for increasing size, performing almost as well as ALGOPERFECT (which is the lower bound for latency in this case) and manages to maintain a near 100% cache hit ratio, especially for smaller sampling rates. Another observation is that the performance of ALGOPERFECT is still significant – this means that there are several constant-time over-heads, which could be further optimized, allowing DICE to perform even faster. Figure 10 performs the same experiment at a larger scale on CLUSTERCLOUD, allowing for cube exploration over the 1 billion rows (100% sampling) while maintaining a sub-second average latency – 33% faster than the baseline ALGONOSPEC.
3.9.4.2 Sampling and Accuracy

Since DICE allows the user to vary the sampling rate, we present a plot of the AVERAGE ACCURACY for a sample workload, compared to results from aggregation over the full dataset. It should be noted that accuracy depends on multiple factors. First and foremost, accuracy is dependent on skew in the data. As described in the schema, our dataset contains a multitude of distributions across all the dimensions. Second, the selectivity of queries in the workload will impact the sensitivity of error. Third, the placement of the data is a significant contributing factor: since data is horizontally sharded across multiple nodes, sampling and aggregation of data is impacted by the uniformity of data placement. In Figure 11, we present the average accuracy for a workload at varying sampling rates over all 1B rows. For this workload, accuracy increases steadily till the 50% mark, after which the benefits of increasing the sampling taper off, slowly reaching full accuracy at the 100% sampling rate.

Figure 11: Accuracy over a workload
3.9.4.3 Number of Dimensions

Figure 12: Impact of Number of Dimensions

Figure 12 shows how varying the number of dimensions in a query affects its execution time. Dimensions are increased by adding new WHERE predicates to the query. As seen in Figure 12, execution time decreases up to a certain point and then starts increasing. The decreasing slope in the curve is caused by selectivity – as dimensions are added, less number of rows are processed, allowing for faster materialization of resultsets. After a certain point, the evaluation cost of the multiple WHERE clauses takes over, especially because the order of filter dimensions is not ideal.
3.9.4.4 Number of Slave Nodes

We vary the number of slave nodes in Figure 13, while keeping the size of the data constant at 200M rows. As expected, for all algorithms, latencies decrease as the number of nodes increase. An interesting observation is made for ALGODICE however – for 4 nodes, DICE thrashes memory due to the amount of data involved and the number of speculative queries, which is not a problem for both ALGONOSPEC (no speculation / caching) or ALGOPERFECT (exactly one ad-hoc query being cached).

Figure 13: Varying the Number of Slave Nodes
3.9.4.5 Cache Hit Variability

Since the cache hit rate is a key contributor to the average latency of a session, in Figure 14 we study how the cache hit rate varies with the sampling rate of a dataset for a fixed cache size. We measure the prediction quality as a proportional measure of the cache hit rate. Higher cache hits are a direct result of high quality of speculation. The experiment result is calculated as average across three different workloads each having 10 queries, running 3 iterations each. We achieve close to a 100% hit rate for 50 million sampled rows. As we increase the sampling rate, we see that the cache hit rate decreasing nearly linearly, since there are more queries to sample and there is a finite number of speculative queries that can be run within the speculation window.
3.9.4.6 Sample Session

As an anecdotal example, we present in Figure 15 the trace of a single cube exploration session for ALGONOSPEC, ALGODICE and ALGOPERFECT on CLUSTERSMALL. The X-axis depicts successive ad-hoc queries in a session. (It should be noted that while the bars are stacked together for convenience to the reader, the session for each algorithm is executed separately.) The Y-axis represents AVERAGE LATENCY. Cache hit rate for ALGODICE is shown as a label above the bars. The cache hit rate for the first query (i.e., the first group of 3 bars) is 0.0, since there has been no speculation and the caches are empty. Since ALGONOSPEC does not speculate any queries throughout the session, the hit rate is always 0.0. Since ALGOPERFECT perfectly speculates each successive query, the results to each successive query is in cache. Thus, all but the first of its queries have a cache hit rate of 1.0. This contributes to the significantly shorter than the corresponding

Figure 15: Individual Latencies for Anecdotal Query Session: NOSPEC, DICE and Perfect Algorithms
bars (i.e. latency for the exact same query) than ALGONOSPEC. ALGODICE, which predicts the forthcoming queries, performs almost as well as ALGOPERFECT with hit rates equal or closer to 1.0.

### 3.9.4.7 Impact of Various Techniques

We now study in Figure 16 the performance impact of the various algorithms and optimizations to our system on the CLUSTER$_{SMALL}$ cluster. We compare the AVERAGE LATENCY of various techniques compared to ALGONOSPEC.

**Figure 16: Impact of Various Techniques**

ALGOUNIFORM is slightly faster due to some of the speculative queries being part of the session. Including the unification optimization discussed in Section 3.8 reduces the number of concurrent queries, contributing to an improvement in latency. Finally, including the locality model and cost-based prioritization of speculative queries yields ALGODICE, which outperforms all other methods.
3.9.4.8 Tuning Parameter: Table Shard Size

Figure 17: Varying the Table Shard Size

Figure 17 shows the effects of varying the shard sizes on the execution time given fixed overall data size of 200M rows of DATAPRIVATE on the CLUSTERPRIVATE cluster. The graph shows 3 varying shard sizes of 500K, 1M and 10M rows per shard. At 500K rows per shard 200M rows of data across 15 nodes equates to 27 shards per node having 4 workers per slave node, this results in 7 queries that need to be served per worker within the short time span of the \( \tau_Y \). This results in a significant context switching at the slave nodes due to the concurrency / master-slave communication to transmit the results, which adds to the latency. This condition is not seen for shard size of 1M, since it results in considerably less number of shards and less context switching. On the other side of the scale, having shards of size 10M results in a lack of sufficient concurrency. This is visible in \textsc{ALGO\_NOSPEC} numbers. \textsc{ALGODICE} latency is high in part because of the overhead of executing incorrect speculative queries. For every wrong speculation, the query has to
be executed and the result sequentially materialized. This results in lesser cache hits and hence more latency. Thus we observe that 1M shard size gives us an ideal tradeoff between context switching, concurrency and the cost of executing a wrong speculation.
Chapter 4: Interface for Cube Exploration

In the previous chapter we introduced DICE, a new way to interactively interact with large data cubes. In this chapter we extend the DICE system to provide a frontend for interactive cube exploration. The frontend supports two modes of operations, an offline mode and an online mode. The offline mode primarily focuses on full cube materialization, which we discussed in Chapter 2. While the offline mode is important for fully materializing the cube, it is not well suited for cube exploration in an interactive manner. DICE is an alternative to typical offline cube materialization and exploration strategy that facilitates on-demand exploration of data cubes at interactive speeds using our novel faceted cube exploration model as outline in Chapter 3. With this new cube exploration model, users can now explore the entire cube lattice without actually typing ad-hoc queries. The DICE system automatically suggests all possible traversals in cube lattice based on the current view/facet in the lattice thereby facilitating faceted cube exploration. The frontend also supports an interactive query editor for advanced users where ad-hoc queries can be explicitly typed.

The following sections will cover the query flow from frontend interface to backend execution engine and also describes various features of the frontend.
Figure 18: Snapshot of DICE Frontend

4.1 Query Flow

Given below is the overall query flow of DICE in terms of the sequence in which the user query, accuracy queries and the speculative queries are scheduled and executed.

The user query is first executed. As soon as the results are returned, the same user query is run with the additional measures of count, sum and variance which are needed to estimate the combined sampling accuracy of measure in the case that the measure is mean or sum. At the same time, the results for the user query are aggregated. It is possible to run the query with the additional measures added to the user requested query itself. However, this results in around 20% more time for the query to return due to the additional measures. By running the accuracy query when the user query returns, we
present the results to the user faster. In the case where the results are presented with sub-second latency, the time a user would peruse through the results would be orders of magnitude greater than the query execution time. Also the time that a user spends in waiting for the result has no benefits. Hence, in our model we choose to run the accuracy query after the completion of actual user query and provide the error bars after a delay.

The delay is also masked by the fact that it overlaps with the time the user is viewing the result. As soon as the accuracy is computed and returned back to the user, the speculative queries are run. The sequences of query executions are illustrated in Figure 19.

Figure 19: Query Flow


4.2 The DICE User Interface

The DICE frontend as shown in Figure 18 has two modes – an offline mode and an online mode. In the offline mode, upon specifying the dimensions the user wishes to cube on and the destination table, the system computes the entire cube at 100% sampling rate and stores it. All the future queries are answered by looking up the materialized data cube. Clearly, for large datasets, the full cube materialization strategy may take a lot of time and is not practical.

In the online mode, the user specifies the query she wishes to execute and the corresponding sampling rate. DICE returns the results for each facet group, additionally reporting sampling errors. The user experience for the online cube exploration depends on the following interface components:

4.2.1 Configuration

In the top left corner of the UI, a user can specify the sampling rate, the choice whether to use speculative execution (Algorithm: DICE) or not (NOSPEC), and establish schema hierarchies upon the attributes in the fact table at hand.

4.2.2 Lattice Explorer

As the user inputs the dimensions and schema, the hierarchy-aware lattice is computed on the fly and displayed, allowing the user to get a feel for the data cube. Hovering over a region highlights the possible traversals in the lattice (sibling and pivot traversals lie
within the same region and are highlighted as the hovered region itself). Clicking on the region in the lattice will result in a popover dialog showing all the valid facet traversals. In case of a sibling, pivot or child traversal, a pop-up prompts the user to choose a value of the selected WHERE clause. On selecting the value, the UI executes the corresponding query and updates the user interface. Thus, the Lattice Explorer helps the user perform faceted cube exploration by simply pointing and clicking.

### 4.2.3 Interactive Query Editor

One way to use the query editor is to simply to write the SQL query in the textbox provided. The alternative is to double click on either the WHERE clauses, GROUP BY keyword or the grouping dimension, which triggers the aforementioned popover, allowing the user to edit the query without advanced knowledge of the query language, schema or data. The options presented by double clicking on the WHERE keyword are to add a dimension resulting in a child traversal, removing a WHERE predicate causing a parent traversal, or a sibling traversal in case of a change in the value. In the case that the WHERE predicate is to be changed, the system will prompt the user with the range of values for each dimension. Similarly, double clicking a WHERE clause also allows the user to remove the dimension, resulting in a parent traversal, or to change the value, resulting into a sibling one. Double clicking on the grouping dimension or the GROUP BY keyword prompts a pivot traversal. After the changes have been made to the SQL query, the query is automatically executed. Thus, the Interactive Query Editor allows the user to perform a full faceted cube exploration by providing with different traversal
options in a visual manner without having to type any part of the SQL query. This intuitive interface has been shown to be really helpful to non-expert users in preliminary informal user studies. Thus, both the Lattice Explorer and the Interactive Query Editor help the user navigate the entire data cube in a series of clicks. It is up to the user to use the more visual nature of the Lattice Explorer or participate in a more hands-on fashion with the user-assisting inputs from the Interactive Query Editor.

4.2.4 Execute

Below the editor is an Execute button, which will need to be clicked in case the query is manually entered into the Query Editor. In case the Lattice Explorer or the interactive features of the editor are used, there is no need for the Execute button to be clicked.

4.2.5 Timing Statistics

As a diagnostic aid, DICE also shows response time, latency, cache hit rate, and a breakdown of the execution time between the backend execution, network transfer and result aggregation. This helps provide feedback about how well the system is able to model user behavior.

4.2.6 Histogram Viewer

Results of the aggregation are presented at the bottom. The behavior of the viewer is such that it displays a scatter plot as soon as possible to the user, maximizing utility. Since queries are typically performed over sampled data, the system issues a secondary set of queries to infer the error estimates, which it overlays as a next step, by adding error bars.
The user views the result and then performs the next set of actions as part of the cube traversal.

### 4.3 List of Possibilities

*DICE* frontend can further be extended with the following features:

- Auto-completion of schema hierarchies using metadata from master node.
- Pan and zoom support in histogram viewer for large data visualizations.
- Drilling down by selecting values from histogram viewer.
- Auto-suggestion of predicate values while typing queries in interactive editor.
- Adding rich interactive elements to editor as discussed in [43].

### 4.4 Video Demonstration

A video demonstrating the usage and features of *DICE* is available at [http://www.cse.ohio-state.edu/db/vldb2013/dice](http://www.cse.ohio-state.edu/db/vldb2013/dice)
Chapter 5: Related Work

Cube Exploration: There exists a significant body of work in data cube exploration. While the original cube paper [12] provides for a variety of operators on the data cube, the facet traversals introduced in this paper correlate directly to interactions on existing user interfaces of business tools such as Tableau. Work by Sarawagi et al. in the mining of interesting cube regions [30], exposing operators that further enable exploration [31] and entropy-based methods of exposing insights are complementary to the ad-hoc exploration our work focuses on. Kamber et al. [19] have discussed the use of metarules and rule mining towards the exploration of interesting cube groups. Architectural solutions to dynamic exploration of cubes by slicing and dicing on subsets of cubes have been discussed in [21]. In contrast, our contribution is towards improving interactive exploration in a session context. The introduction of such exploration methods into our framework is ideal future work.

Cube Materialization: Distributed materialization strategies that support ad-hoc interactive exploration range from full-cube materialization over MapReduce [24] to region-specific materialization [8] to selective partial materialization trading approximation and query execution time given space constraints. Optimization techniques exist for optimizing intra-query parallelization [3], but do not consider
multiple queries as part of an interactive session. Combining partial, offline materialization and the online cube exploration ideas proposed in this paper would make for an ideal practical implementation.

**Distributed Query Execution:** Ad-hoc analysis over large datasets has been made popular with the availability of declarative query languages such as SCOPE [6], Pig [27] and Hive [34], which translate to batch-oriented MapReduce-oriented flows, where the entire query pipeline has to finish executing before any results are surfaced. While this approach allows for ad-hoc large-scale distributed query execution, it is not ideal for interactive workloads. Optimizations such as columnar storage layouts [13], hierarchical execution [22], distributed database hybrids [1] and main-memory engines [11] have demonstrated the ability to achieve low latencies when querying over large datasets.

These ideas have resulted in a spurt of development activity in this area, resulting in implementations such as Drill, Impala, Tez, PivotalHD, HAWQ, Peregrine and Druid. It should be noted that each of these projects target *single query execution latency*. Ideas presented in *DICE* and these projects can be used in a complementarily fashion to further improve the session-based interactive cube exploration performance.

**Prefetching:** The idea of speculative execution of queries and the prefetching of result sets has been discussed before [33]. PROMISE [29] investigates the likelihood of future queries and can be used to supplant the workload-based approach in our paper. Ramachandran et al. [28] focus on the speculation of exact, non-approximate drill-down queries. Improvements in speculation quality based on ideas in these papers can be used
to better prioritize and sample our speculative queries, allowing for further improvement in the interactivity of cube exploration sessions.

*Online Aggregation:* Our work is inspired by online aggregation ideas proposed by Hellerstein et al. [17] and the related CONTROL [16] project, which surface approximate answers to the user in an online fashion using sampling-based techniques for interactivity. Our system extends these ideas to a distributed environment, focusing on the exploration of data cubes. Our core contribution of a framework to combine speculative execution and sampling based approaches benefits from online aggregation ideas.

*Sampling-based Estimation:* There is significant prior work [25] in sampling-based estimation methods for approximating query results over large and distributed datasets. Jin et al. [18] detail the approximation of OLAP queries using summary statistics that are built in an initial pass over the entire data. Wang et al. discuss [37] data placement as a key contributor to platforms with analytical workloads. BlinkDB [2] performs an offline sampling step of multiple column combinations, allowing for biased sampling during the online query step. The use of stratified sampling is especially ideal in distributed contexts, allowing for the execution of strata on different nodes followed by an aggregation step. This approach is closest to our use of sampling, and the ideas presented in BlinkDB are orthogonal to both the faceted exploration model proposed by our work, and the speculation-based execution architecture. As described in the evaluation section, our non-speculation baseline is our distributed, sampling-based execution, which can be
compared to BlinkDB. Strategies for stratification using prior workloads [7] and methods to increase sensitivity for low-selectivity attributes [36] have been considered. We leverage similar insights in the prioritization of speculative queries (described in Section 3.4), and using prior workloads can be done trivially by including the prior query frequency as a weight.

**Data Interaction:** The proliferation of business intelligence tools that leverage visualization and interactive interfaces [38, 4, 9, 20] to explore large multidimensional datasets highly motivate the need for a distributed interactive cube exploration system. Tools such as Tableau [14] translate visual interactions into a series of SQL queries, and interactive loops correlate directly with our session-based model. As discussed in Section 4, we observe that such interactions directly correspond to facet traversals, allowing us to utilize actual workloads from such tools in our experimental evaluation. Olston et al. [26], propose the interactive analysis of web-scale data using query templates, while Cetintemel et al. [5] provide a vision for a “guidance” system for interactive querying. The session-oriented sampling and speculation approach described in our paper can significantly improve the interactivity of such a system.
Chapter 6: Conclusion and Future Work

We introduced *DICE*: a distributed, interactive cube exploration system. Given the proliferation of commodity distributed infrastructure and big data analytics applications, there is a compelling need for systems that allow interactive exploration of aggregated data. As demonstrated in the experiments, *DICE* meets this need, and allows for exploration of 1 billion tuple data cubes at sub-second latencies, significantly outperforming existing methods. The system uses a combination of three complementary strategies: a faceted cube exploration model, data sampling and speculative caching to provide interaction-level performances for the end-user.

Going forward, there are several avenues of future work. The inclusion of *interestingness* of cube groups into the exploration framework would be a very useful extension. This would bridge the gap between automated exposition of insights, and ad-hoc exploration. One possible way to include this into *DICE* is to formulate the *interestingness* of a facet, which can be mined in an initial offline step during the ingestion of a table shard, and stored in conjunction with the dataset. During the exploration phase, the interestingness score is considered in addition to the probability and gain values when prioritizing facets to speculatively execute.

Another possible extension is to combine methods for offline and online materialization.
of data cubes. Identifying the fraction of the cube to fully pre-materialize, and the portion to rely on ad-hoc execution is an interesting problem. Given space constraints, a possible approach is to materialize an approximate and compressed representation [10] of the data cube, and use the online execution step to increase the quality of the answer based on the approximate model [15].
References


Appendix A: Combination of Variances

We describe below how the variances of the estimators of measures for the same group obtained by running the same query on multiple shards can be combined

\[ s^2 = \frac{1}{n-1} \sum_{i,j} (x_{ij} - m)^2 \]

\[ = \frac{1}{n-1} \sum_i \sum_j (x_{ij} - m_i + m_i - m)^2 \]

\[ = \frac{1}{n-1} \sum_i \sum_j \left( (x_{ij} - m_i)^2 + (m_i - m)^2 + 2(x_{ij} - m_i)(m_i - m) \right) \]

\[ = \frac{1}{n-1} \left( \sum_{i,j} (x_{ij} - m_i)^2 + \sum_i n_i (m_i - m)^2 + 2 \sum_i (m_i - m) \sum_j (x_{ij} - m_i) \right) \sum_j (x_{ij} - m_i) \]

\[ = \frac{1}{n-1} \left( \sum_i (n_i - 1)v_i + \sum_i n_i (m_i - m)^2 \right) \]

where for a particular group, \( n_i, m_i, v_i, x_{ij} \) are respectively the number of tuples, the mean, the variance, the \( j^{th} \) observation of the results from the \( i^{th} \) shard; \( n \) is the total number of tuples belonging to the group, \( m \) is the overall mean of the group.