Sampling-based Techniques for Interactive
Exploration of Large Datasets

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor
of Philosophy in the Graduate School of The Ohio State University

By

Niranjan Kamat, B.Tech., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2018

Dissertation Committee:

Arnab Nandi, Advisor
Srinivasan Parthasarathy
Spyros Blanas
Ozeas S. Costa
© Copyright by

Niranjan Kamat

2018
Abstract

Extracting useful information from large datasets has become increasingly important. While our computational capabilities have improved greatly in the last few decades, data explosion has resulted in the growth rate of data vastly out-pacing that of computational power. In such scenarios, obtaining results within interactive (few seconds at the most) response times is extremely difficult. Exploring data interactively has numerous advantages such as shortening the feedback loop, providing the ability to perform numerous experiments, and giving a smoother user experience. As a result, providing results using a sample of the data to improve query response time has become hugely important.

In this work, I will present our efforts in the direction of interactive exploration of large-scale datasets within interactive response times with projects such as DICE, Sesame, FluxQuery, and a unified join sampling approach. We use approaches such as speculative execution, data sampling, faceted exploration, and scan sharing towards this end.

In an OLAP scenario, we note that queries occur not in isolation but as part of a larger query session, where queries might be similar to the previously executed queries. DICE uses this session-oriented behavior of a user to speculatively execute and cache the likely follow-up queries, so that the user query can be answered quickly from the cache. DICE scales up to a billion tuples within sub-second response time using 50 nodes. Further, DICE has a novel, intuitive, synergistic interface designed to aid the speculation and faceted exploration used by the backend.
In the context of sampled aggregations, the results provided to the user have lesser significance if not provided with the corresponding error bars as well, which depend on the variance of the measure. Variance can be expensive to compute. In Sesame, we investigate different techniques for reusing variance computations in order to speed-up error delivery. We use the entire user query session to build an optimal set of speculative queries to run, giving results alongside errors nearly 25× faster.

Supporting sampling in the presence of joins is an important problem in data analysis, but is inherently challenging due to the need to avoid correlation between output tuples. To sample joins, we present a unified approach through two key contributions. First, in the case where a correlated sample is acceptable, we provide techniques, for all join types, to sample base relations so that their join is as random as possible. Second, in the case where a correlated sample is not acceptable, we provide enhancements to the state-of-the-art algorithms to reduce their execution time and intermediate data size.

Modern computing devices and user interfaces have necessitated highly interactive querying. Some of these interfaces issue a large number of dynamically changing queries to the backend. In FluxQuery, we propose a novel model to interpret the variability of likely queries in a workload. We implement a cyclic scan-based approach to process queries from such workloads in an efficient and practical manner, while reducing the overall system load.
Dedicated to my wonderful daughter Sia
Acknowledgments

I would like to profoundly thank Arnab for his guidance and all the help he has provided. His insight, innovative thinking, and push for me to excel has immensely improved my skill-set. I will always owe a whole lot of my success to Arnab. He has been a perfect advisor and has helped me accomplish all goals I had set before starting my PhD.

Srini and Spyros have been perfect committee members and collaborators and cannot thank them enough for their guidance. Kannan Srinivasan has also provided invaluable comments during the dissertation process. Eugene Wu has helped me a lot on multiple projects as well. I would also like to thank Dr. Ozeas S. Costa for his time and effort spent being a member of my dissertation committee.

I am also greatly thankful to Lilong Jiang. I could not have wished for a better friend.

I have enjoyed not only working with Roee Ebenstein, but also the endless mindless conversations.

I would also like to thank Karthik Tunga and Prasanth Jayachandran for an amazing friendship and collaboration on DICE. The same goes for the entire data team gang – Tarun Sasikumar, Vineet Nair, Farhang Zia, Chaitanya Solapurkar, and Ramiya Venkatchalam.

Ziqi Huang, Meraj Khan, Protiva Rahman, Behrooz Omidvar-Tehrani, Ritesh Sarkhel, David Palzer, and Ben Strauss have provided great help and have been ideal colleagues these past few years.

I am very thankful to my family, especially my amazing daughter Sia.
Vita

February 1985 .................................. Born,
Mumbai, India.

June 2007 ................................. B.Tech.,
Computer Science,
VNIT, Nagpur, India.

June 2010 ................................. M.S.,
Computer Science,
University at Buffalo, SUNY.

December 2017 ............................... M.S.,
Computer Science,
The Ohio State University.

Summer 2014 ................................. Software Engineering Intern,
Google.

Summer 2017 ................................. Software Engineering Intern,
Amazon.

Publications

Research Publications


**Fields of Study**

Major Field: Computer Science and Engineering
Table of Contents

Abstract ................................................................. ii

Dedication .............................................................. iv

Acknowledgments ...................................................... v

Vita ............................................................................. vi

List of Tables ........................................................... xiii

List of Figures ........................................................... xiv

1. INTRODUCTION ....................................................... 1
   1.1 Motivation: Need for Interactivity ............................... 1
   1.2 Traditional Approaches ........................................... 2
      1.2.1 Data Cubing .................................................... 2
      1.2.2 Sampling & Online Aggregation ......................... 3
   1.3 Motivating Example ................................................ 4
   1.4 Problem Statement ................................................ 5
   1.5 Proposed Solution ................................................ 5
      1.5.1 Speculation & Caching ..................................... 6
      1.5.2 Sampling The Large Denormalized Fact Table ........ 6
      1.5.3 Extending Reuse to Error Computation ............... 7
      1.5.4 Co-designing User Interface with DICE backend ..... 7
      1.5.5 Sampling Joins between Tables ......................... 7
      1.5.6 Execution Engine for In-flux Queries ................ 7
   1.6 Contributions ..................................................... 8
      1.6.1 DICE: Distributed Interactive Cube Exploration .... 8
      1.6.2 Sesame: A Session-Aware Approach to Reusing Error Computations in Sampled Aggregations 9
1.6.3 Co-designing *DICE* User Interface with Session-based Backend ........................................ 10
1.6.4 A Unified Correlation-based Approach to Sampling Over Joins .................................... 10
1.6.5 FluxQuery: An Execution Framework for Highly Interactive Query Workloads .................. 11
1.7 Comparison with State-of-the-Art ......................................................................................... 11
1.8 Organization ....................................................................................................................... 12

2. Distributed and Interactive Cube Exploration ........................................................................... 13

2.1 Introduction .......................................................................................................................... 13
2.2 Data Model and Preliminaries ............................................................................................. 14
  2.2.1 Faceted Exploration of Data Cubes .............................................................................. 14
  2.2.2 Distributed Execution .................................................................................................. 18
  2.2.3 Querying over Table Shards ....................................................................................... 19
2.3 Variance Combination .......................................................................................................... 21
2.4 The *DICE* System ............................................................................................................. 23
  2.4.1 Speculating Queries in a Session ............................................................................... 23
  2.4.2 System Architecture ................................................................................................... 25
  2.4.3 Query Flow ................................................................................................................ 26
  2.4.4 Prioritizing Speculative Queries ............................................................................... 27
  2.4.5 The *DICE* Algorithm .............................................................................................. 29
  2.4.6 Optimization: Query Unification ............................................................................... 30
  2.4.7 Optimality of *DICE* ................................................................................................. 32
2.5 Experiments & Evaluation .................................................................................................... 33
  2.5.1 Experimental Setup ................................................................................................... 33
  2.5.2 Results ....................................................................................................................... 35
  2.5.3 Real-world Usage and User Study ............................................................................ 40
2.6 Related Work ....................................................................................................................... 42
2.7 Conclusion & Future Work ................................................................................................... 44

3. A Session-Based Approach to Fast-But-Approximate Interactive Data Cube Exploration .......... 45

3.1 Introduction .......................................................................................................................... 45
3.2 The *Sesame* Model ........................................................................................................... 47
  3.2.1 Query Model ................................................................................................................ 47
  3.2.2 Session Model .............................................................................................................. 48
3.3 Error Computation in *Sesame* .......................................................................................... 48
  3.3.1 Error Reuse ................................................................................................................ 49
  3.3.2 Efficiency of Variance Calculations .......................................................................... 51
  3.3.3 Extensibility to Other Measures ............................................................................... 51
3.4 The *Sesame* Framework .................................................................................................... 52
7. Conclusion ................................................................. 149
  7.1 Contributions ......................................................... 149
  7.2 Limitations of Sampling ........................................... 149
  7.3 Applicability of Sampling ......................................... 150
  7.4 Test of Time ........................................................ 151
  7.5 Future Work ........................................................ 151

Bibliography ................................................................. 153
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Techniques employed by our systems.</td>
<td>8</td>
</tr>
<tr>
<td>2.1 List of notations used in this section.</td>
<td>21</td>
</tr>
<tr>
<td>5.1 List of Notations</td>
<td>89</td>
</tr>
<tr>
<td>5.2 Effect of Sample Allocation on Number of Samples</td>
<td>90</td>
</tr>
<tr>
<td>5.3 Error in Single Stratum Partitioning</td>
<td>101</td>
</tr>
<tr>
<td>5.4 Error in Multiple Strata Partitioning</td>
<td>102</td>
</tr>
<tr>
<td>6.1 Interactivity Over Data(<em>{100MB \times 10MB}), Machine(</em>{4GB}) for simultaneous queries (threads). NL, CNL, FJ, and FFJ are the only interactive algorithms in this case.</td>
<td>141</td>
</tr>
<tr>
<td>6.2 Interactivity Over Data(<em>{1GB \times 1GB}), Machine(</em>{4GB}) for simultaneous queries (threads). Here, NL, FJ, and FFJ are the only interactive algorithms.</td>
<td>142</td>
</tr>
<tr>
<td>6.3 Interactivity Over Data(<em>{100GB \times 100GB}), Machine(</em>{244GB}) for simultaneous queries (threads). Here, CNL, FJ, and FFJ are the only algorithms that scale for large relations, while being interactive.</td>
<td>142</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td><em>DICE</em> Approach</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>In order to execute queries over large datasets within interactive response times, we use sampling to reduce the amount of data to be processed and speculation to execute likely future queries in order to answer the user query from the cache.</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Faceted Cube Exploration Traversals</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>System Architecture</td>
<td>25</td>
</tr>
<tr>
<td>2.3</td>
<td>Varying Size of Dataset (<em>CLUSTERSMALL</em>)</td>
<td>36</td>
</tr>
<tr>
<td>2.4</td>
<td>Varying Size of Dataset (<em>CLUSTERCLOUD</em>)</td>
<td>37</td>
</tr>
<tr>
<td>2.5</td>
<td>Accuracy over a workload</td>
<td>37</td>
</tr>
<tr>
<td>2.6</td>
<td>Varying the Number of Worker Nodes</td>
<td>38</td>
</tr>
<tr>
<td>2.7</td>
<td>Cache Hit Change with Sampling Rate Change</td>
<td>39</td>
</tr>
<tr>
<td>2.8</td>
<td>Individual Latencies for a Query Session</td>
<td>39</td>
</tr>
<tr>
<td>3.1</td>
<td>Common Error Visualizers: The figure depicts a sampled aggregation query with some commonly used representations to visualize errors. It shows <em>Error</em> estimates (often represented visually as error bars), which are an important part of the output of <em>sampled aggregation</em> queries. <em>ERR()</em> represents error estimates such as standard deviation, standard error, or other variance-related computations that are expensive to compute.</td>
<td>46</td>
</tr>
</tbody>
</table>
3.2 Query Workload and Reuse Model ........................................... 47
3.3 Sesame System Architecture .................................................. 53
3.4 Different Dataset Sizes ($Workload_{TPCDS}$) ................................ 62
3.5 Different Dataset Sizes ($Workload_{Real}$) .................................... 63
3.6 Overall Speculation Benefit ..................................................... 64
3.7 Effect of Speculation Duration on Response Time ......................... 64
3.8 Effect of Speculation Duration on Cache Hit Rate ......................... 65
3.9 Comparison of View Sizes ...................................................... 66
3.10 Comparison of $Variance_{Fast}$ and $Variance_{Common}$ ....................... 67
3.11 Distribution of Execution Time in Sampled Aggregations ................. 68
3.12 Naive Aggregation vs Aggregation Tree .................................... 69
3.13 Effect of Increasing Number of Cores ($Workload_{Real}$) .................... 70
3.14 Effect of Multiple Users ($Workload_{Real}$) ................................... 71
3.15 Sesame on Impala ($Workload_{Real}$) .......................................... 72
3.16 Comparison against BlinkDB ($Workload_{Real}$) ............................ 73
4.1 DICE User Interface Query Flow ............................................. 79
4.2 DICE User Interface that allows for interactive, intuitive cube exploration. 81
5.1 Stratified Random results in the most possible samples. MAXRANDJOIN always results in more samples than Ripple Join, while Universe Sampler results in the fewest. ............................................. 104
5.2 MAXRANDJOIN has the least error amongst correlated samplers. Interestingly, STRATJOIN has lesser error than MAXRANDJOIN for the smallest sample size. ............................................. 105
5.3 MAXRANDJOIN is resistant to white Gaussian noise with low standard deviation. At higher sample sizes, it can handle binomial perturbations to join count information. ............................................. 105

5.4 While an increase in the sampling rate results in a linear increase in the intermediate data size, the size required by GROUP-SAMPLE-ENHANCED is multiple orders of magnitude smaller than that of Group-Sample due to the usage of reservoir sampling-based techniques. ...................... 107

5.5 GROUP-SAMPLE-ENHANCED intermediate data size depends on the join size, while Group-Sample sample size does not and can increase regardless (LHS skew = 0). ...................................................... 107

5.6 GROUP-SAMPLE-ENHANCED provides results around an order of magnitude sooner. It is also able to finish execution within the time limit in one of the cases where Group-Sample is not (f = 1%). ......................... 108

5.7 GROUP-SAMPLE-ENHANCED handles increasing skew better as well. .... 108

6.1 Exploratory Join Example [106] ......................................................... 124

6.2 Query intent transition comprising 4 queries. ................................. 127

6.3 FluxQuery Architecture ................................................................. 129

6.4 Time To Completion, Data_{100MB \times 10MB}, Machine_{4GB}. ............ 140

6.5 Time To Completion, Data_{1GB \times 1GB}, Machine_{4GB}. ................. 140

6.6 Join Types Comparison, Data_{10MB \times 10MB}, Machine_{4GB}. Here, FJ and FFJ represent a good tradeoff between CHJ (non-interactive) and CNL (interactive). 143

6.7 Join Types Comparison, Data_{1024MB \times 100MB} on Machine_{244GB}. Here, FFJ is ideal for interactivity. ................................. 144
Chapter 1: INTRODUCTION

Large-scale analytics has found a growing number of use cases in a variety of disciplines, from business to the sciences. With the rapid rise in the size of available data, and reliance on data-driven insights for decision making, planning, and analysis, the role of analytics over massive datasets has become a critical one. Modern database architectures often use parallelism with multi-threaded or multi-core solutions to reduce execution times towards this end [138].

With the 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 dedicated and dynamically provisioned distributed computational resources allows analyses that were typically handled by database administrators to be performed by the end-users of the analyses themselves.

1.1 Motivation: Need for Interactivity

There is an increasing demand for 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. Data analysis is often expected to be interactive – queries need to be responded to within a small latency bound. Studies conducted in the Human-Computer Interaction domain [95, 103, 132] have heavily motivated a sub-1000 ms limit for query
response time: the minimum time for the system to respond with a result without impeding the user’s interaction flow. They have established guidelines and demonstrated the functional and economic value of rapid response times – empirical tests on our system have shown the same. However, providing results within interactive response time is not always possible as the data size increases, despite the availability of performant, distributed, and scalable infrastructure.

1.2 Traditional Approaches

1.2.1 Data Cubing

Intuitively, the simplest approach to ensuring fast, interactive analytics is to materialize the entire Data Cube [57] so that each query to the cube is simply a lookup from the main memory cache. The data cube can trivially be explained as the union of all possible combinations of GROUP BY queries for different desired aggregate measures. Such a representation is useful for the purpose of exploratory data analysis since successive investigatory questions can be answered in the form of drill-down or roll-up queries. 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, which typically is larger than available memory [65]. Hence, numerous techniques have been developed for partial materialization of the cube [64, 65]. However, 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. Further, the cube needs to be updated upon changes to the underlying data. However, this is not always possible – the minimum value might be deleted in case of the measure MIN or any tuple might be deleted in the case of MEDIAN. In such cases, we can see that offline cube materialization is not helpful.

1.2.2 Sampling & Online Aggregation

In a wide variety of domains and use cases, a user is happy to trade-off accuracy for faster query response times. Sampling is particularly applicable in all domains that rely on data exploration. Accuracy-critical applications such as healthcare and finance can also be benefited by sampling as a preliminary analysis, followed by exact computation (Sections 7.2 and 7.3 discuss limitations and applicability of sampling in detail). Thus, an often-used approach is to execute the query over a sample of the data, computed offline [4, 30, 92, 150]. However, this approach cannot accommodate changes in the underlying data. The techniques described in our work are complementary to such an approach and can easily be adapted, if needed, to accommodate offline sampling. Online aggregation builds upon sampling by providing query results upon the start of the query and updating the results as more data is processed [69]. However, online aggregation requires a significant overhaul of the entire query processing infrastructure. Our approaches are influenced by the online aggregation-based techniques. Our work focuses on sampling techniques for using an online sample of randomized data to provide results within interactive response times.
1.3 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 system 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 can be given by:

```sql
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 removing any need for user intervention, but about helping the user analyze the data faster by reducing the time it takes to interact with it.

Our use case is driven primarily by the need for interactive data cube exploration. 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. Another consequence of the size of the data is that it is *impractical to construct a fully materialized cube* to perform the analysis. Interaction between the user and the system needs to be fluid, requiring the underlying queries to return quickly, enforcing the latency bounds discussed above. This necessitates using a sample of the data to answer the query. Lastly, queries are seldom one-off, and almost always occur as part of a larger *session of related queries*.

1.4 Problem Statement

In light of this characterization, our problem statement can be given by:

*Given relations that are to be explored in a session-oriented fashion, and the queries issued by the user so far, which could be varying continuously, ensure that the queries are responded to at the earliest, within user-specified response times.*

We look at this problem from the perspective of the number of relations that are handled – a single relation or multiple relations (joins). This approach is influenced by the standard techniques employed in data warehouses – some propose to create a large denormalized fact table by joining all possible tables, while others propose to keep the tables in their normalized form. DICE and Sesame handle the case of single relations, while our join sampling framework and cyclic scan-based approaches tackle joins.

1.5 Proposed Solution

To solve this problem, we propose using numerous techniques provided below, and summarized in Figure 1.2.
Figure 1.2: In order to execute queries over large datasets within interactive response times, we use sampling to reduce the amount of data to be processed and speculation to execute likely future queries in order to answer the user query from the cache.

1.5.1 Speculation & Caching

As noted above, in an OLAP scenario, queries occur not in isolation but as part of larger sessions. As a result, it is possible to execute and cache likely follow-up queries, so that the next user query can be answered from the cache, which can greatly speed-up the query execution. This requires a model for predicting the next user query, for which we use our faceted model.

1.5.2 Sampling The Large Denormalized Fact Table

For large datasets, it is not always possible to return results within interactive response times. This necessitates providing results using a sample of the data. We focus on the selection of views to both speculatively execute and reuse that help minimize the query execution time.
1.5.3 Extending Reuse to Error Computation

The results have lesser importance if not accompanied by the corresponding error bars. Hence, reuse needs to be extended from simply reusing the results for the measures to reusing the errors as well. We use numerous variance computation techniques in order to expedite error calculation.

1.5.4 Co-designing User Interface with DICE backend

An interface needs to be designed to aid our faceted exploration, speculation, and sampling-based backend to help the user navigate through the session in the form of user-friendly point and click tools. We have co-designed the frontend of DICE with the backend, taking these ideas into account.

1.5.5 Sampling Joins between Tables

Performing sampling is difficult in the case of joins due to the correlation between output tuples. We address this problem by providing a unified correlation-based approach that maximizes join randomness and improves upon the state-of-the-art approaches.

1.5.6 Execution Engine for In-flux Queries

User interaction can lead to queries that change over time. Addressing this use case requires us to rethink the traditional querying paradigm. We introduce a query model designed for such in-flux queries. Our cyclic-scan based execution engine is designed to handle such interactive queries.
1.6 Contributions

Our work can also be looked at through the lens of the systems that we have built (Table 1.1). We have developed the DICE backend and frontend synergistically to provide a fast, fluid user experience. Sesame expands on the ideas presented in DICE to show that reuse is not restricted to simply results but can be extended to error as well. These systems employ both sampling and speculation. FluxQuery targets in-flux queries as a result of user interaction. We also target the important problem of sampling joins by considering a holistic solution based on sample correlation. We summarize the contributions of each of our systems below.

1.6.1 DICE: Distributed Interactive Cube Exploration

We propose DICE, a system that uses a session-oriented approach to data cube exploration. In contrast to existing OLAP systems, our system is designed keeping in mind the user’s flow, surfacing approximate results within interactive latencies. DICE can be summarized with the following four key contributions:

- DICE allows exploration of 1-billion-tuple data cubes at sub-second levels.
• We provide a principled cost-based framework that combines two complementary techniques: speculative query execution and online data sampling to achieve interactive latencies for cube exploration in a distributed framework.

• To bound the space of possible speculative queries, we propose a faceted cube exploration model that considers successive queries as part of a query session.

• We share insights into the design and implementation of our system based on real-world query logs, user studies, and detailed performance evaluations.

### 1.6.2 Sesame: A Session-Aware Approach to Reusing Error Computations in Sampled Aggregations

The overarching goal of Sesame is to accelerate sampled aggregation queries without modifying the underlying database. As shown in our evaluation section, Sesame achieves speedups of up to $25 \times$ using a single-node backend (PostgreSQL) and up to $4 \times$ using a distributed backend (Impala). This is made possible by the following contributions:

• We show that error reuse is an often overlooked and yet expensive component of sampled aggregations, and describe methods to extend result reusability to the computation of variance in the context of sampled aggregation queries.

• We provide a novel technique to optimize execution of speculative queries taking into consideration the user behavior and the queries executed so far in the session.

• We provide a cube lattice-based session cache that expedites the search of reusable results, and helps with cube region-aware allocation of the cache resources. We present and evaluate allocation and reuse strategies that are aware of data sharding, approximation, and the session-based OLAP workload.
1.6.3 Co-designing DICE User Interface with Session-based Backend

We introduce the DICE frontend, which is designed keeping the backend in consideration and provides a smooth user experience. It makes the following contributions:

• Our frontend is designed to facilitate traversal in a session-oriented fashion by providing traversals based on our faceted model.

• Due to challenges of scale and expectations of query response time, it is necessary to aggregate over a sample of the data. As shown in Figure 4.2, a user can manipulate the slider to conveniently run queries at different sampling rates, thereby allowing the user-specified trade-off between accuracy and response time via an interface.

• The user studies showed that the backend is indeed often idle in exploratory OLAP querying – we have an opportunity to speculatively execute queries they might run next and cache them.

• The system provides a user flexibility between choosing the easier point-and-click approach or the more powerful option of entering the commands manually.

1.6.4 A Unified Correlation-based Approach to Sampling Over Joins

Sampling over joins is an important and difficult problem as not only can the result size get extremely large, but naive sampling can result in the sample size exceeding the relation size, causing sampling to be counter-productive. We provide a unified correlation-based approach to handle this problem as follows [79].

• First, we look at techniques to maximize the number of possible samples in correlated sampling under the constraint of fixed sample size, when statistics over the join column are available. We provide strategies for allocating samples to different strata of multiple
relations for different join types, including equi-join, outer join, self-join, non-equi-join, and theta join.

• Our second contribution is in non-correlated sampling, where we provide enhancements to the state-of-the-art algorithms [31]. Our techniques reduce the sample size and the intermediate data size, resulting in the time for performing both sampling and join being reduced as well.

1.6.5 FluxQuery: An Execution Framework for Highly Interactive Query Workloads

FluxQuery is designed for handling workloads issued by modern computing devices [43]. Such workloads consist of constantly varying queries. To handle this scenario, we make the following contributions.

• We introduce a novel query intent model that allows us to properly represent highly interactive query workloads.

• We describe FluxQuery, our framework for interactive query execution, based on a novel cyclic join approach.

• We provide algorithms for scheduling and executing such workloads, including novel join algorithms – FluxJoin and Fast FluxJoin.

• We experimentally demonstrate that unlike traditional databases, our implementation can successfully handle highly interactive query workloads within interactive response times.

1.7 Comparison with State-of-the-Art

Our work on speeding up queries for a single relation, which includes DICE and consequently Sesame, was the first to look at using speculation to solve the problem. The notion of using speculative queries to speed up query sessions did not exist at that time.
Since then, a few works including Forecache [15], AIDE [41], and IDEA [39] have been introduced that use the speculation concept. However, unlike them, our backends can scale to large datasets using both scale-out and scale-up architectures.

Join sampling is a difficult and important problem and consequently, there have existed multiple efforts to solve this problem for at least a couple of decades [2, 31, 83, 91]. However, as our work shows, none of these approaches can be used in every use case without suffering from serious drawbacks. We introduce a unified correlation-based framework that recognizes the inherent hardness of the problem to propose a use case dependent solution.

While querying through interactive devices is now common-place, there does not exist a backend built for handling the resulting in-flux queries. Our work models the behavior of such queries to propose a cyclic scan-based query processing engine.

1.8 Organization

The rest of the chapters are organized as follows. In Section 2, we describe our DICE system. Section 3 shows through Sesame that reuse is not restricted to simple user measures but can be used in error calculation as well. In Section 4, we provide the frontend of DICE, which has been developed synergistically with the backend. Section 5 provides our unified approach to the join sampling problem. Section 6 describes our FluxQuery framework for executing highly varying workloads. Finally, we conclude our work, discuss its applicability and limitations, and provide ideas for future work in Section 7.¹

¹This material is based upon work supported by the National Science Foundation under Grant No. 1422977 and 1527779. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.
Chapter 2: Distributed and Interactive Cube Exploration

2.1 Introduction

DICE considers querying in an OLAP scenario for which we use a session-based query model. In exploratory OLAP scenarios, we have noted that queries occur not in isolation but as part of larger query sessions [127]. It is common for users to pose a query, and then issue follow-up queries based on the insights gleaned, or to query related parts of the data. Iterative, session-based queries also make up a significant fraction of the TPC-DS benchmark [117]. Hence, it is important to look at the query processing paradigm in the context of sessions. In these scenarios, since subsequent queries are dependent on the current user query, providing interactive response times carries even greater importance since it helps the user iterate over the query session faster. Further, since each session query needs user supervision and constant refinement, user’s time is inherently coupled with the query response time. As we will see in the following sections, there are several opportunities that arise from a session-based approach to distributed query execution that can lead to significant reduction in the response time.

While the user is perusing the results, the system can populate the session cache with likely follow-up queries, reducing the query response time. We use operators derived from the data cube model as the basis for query speculation (Section 2.2.1). One way to expedite
expensive computations is to parallelize them. A distributive measure is defined as an aggregate function that can be computed in a distributed way, and an algebraic measure is defined as an aggregation function that has a bounded number of arguments, each of which is obtained by applying a distributive function [99]. Hence, the calculation of distributive and algebraic measures can be made highly parallel as well as reusable, resulting in tremendous speedups.

2.2 Data Model and Preliminaries

Having motivated the problem setting of a distributed, interactive, cube exploration system, we now discuss preliminaries for each of these three contexts. We begin with cube exploration, where we define a faceted exploration model to facilitate complete yet efficient exploration of the data cube. As we will discuss in the following section, faceted exploration bounds the space of successive queries, thereby making speculative query execution feasible. Second, we discuss the execution of faceted queries in a distributed setting, where data is distributed across nodes as table shards. Finally, given the constraints of interactivity, we explain our techniques for approximate querying over sampled data, provide a framework to execute faceted queries over multiple nodes, and draw from concepts of stratified sampling to aggregate results and estimate error bounds.

2.2.1 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 [57]. A region denotes a node in the cube lattice and a group denotes tuples with the same values of attributes for that region. For example, one of the groups in the region \{datacenter, month\} is \{EU, January\} for the cube derived from the motivating example. We continue with 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:

\[\text{location}[:\text{zone}:\text{datacenter}:\text{rack}], \text{time}[:\text{month}:\text{week}:\text{hour}], \text{iops}.\]

**Challenges in Exploration:** As a user exploring a data cube, the number of possible parts of the cube to explore 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 [145]. Empirically, most visualizations 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. This successive facet can be a *parent* facet in the case of a roll-up, a *child* facet in the case of a drill-down, a *sibling* facet in the case of change of a dimension value in the group, and a *pivot* facet in the case of a change in the inspected dimension. Thus, the user

![Faceted Cube Exploration Traversals](image)
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 data differently.

A session comprises of multiple traversals. The formal definitions are as follows.

**Facet**: For a region $r$ in cube $C$, a facet $f$ is a set of groups $g \in r(d_1...n)$ such that the group labels differ on exactly one dimension $d_i$, i.e. $\forall g_a, g_b \in f, d_i(g_a) \neq d_i(g_b) \land d_j(g_a) = d_j(g_b)$ where $i \neq j$ and $d_i$ 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 $f(d_g, \overrightarrow{d_b} : v_b)$ where $d_g \cup \overrightarrow{d_b}$ represents a parent group of $\overrightarrow{d_b} : v_b$ in the cube lattice. The parent facet $f(zone, month : m_1)$ generalizes the dimension $time$ from the prior example.

**Facet Session**: A facet session $\vec{F}$ is an ordered list of facets $f_1...n$ 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*, inspired by similar traversals over data cube, 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 $f_p(d_{pg}, \overrightarrow{d_pb} : v_{pb})$ is a parent to the facet $f(d_g, \overrightarrow{d_b} : v_b)$ if $d_{pg} = d_g$ and $\overrightarrow{d_pb} : v_{pb}$ represents a parent group of $\overrightarrow{d_b} : v_b$ in the cube lattice. The parent facet $f(zone, month : m_1)$ generalizes the dimension $time$ from the prior example.

**Child Facet**: A child facet is defined as any facet obtained by specializing any of the bound dimensions. Thus, a facet $f_c(d_{cg}, \overrightarrow{d_cb} : v_{cb})$ is a child to the facet $f(d_g, \overrightarrow{d_b} : v_b)$ if $d_{cg} = d_g$
and $d_{cb} : v_{cb}$ represents a child group of $d_b : v_b$ in the cube lattice. Thus, the child facet $f(zone, month : m_1, week : w_1, hour : h_1)$ specializes the dimension time.

**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 $f_s(d_{sg}, d_{sb} : v_{sb})$ is a sibling to the facet $f(d_g, d_b : v_b)$ if $d_{sg} = d_g$, $d_{sb} = d_b$ and $v_{sb}$ and $v_b$ differ by exactly one value. The sibling facet $f(zone, month : m_1, week : w_2)$ thus changes the value of week.

**Pivot Facet:** A pivot facet is defined as any facet obtained by switching the grouping dimension with a bound dimension. Thus, a facet $f(d_g, d_b : v_b)$ can be pivoted to the facet $f_v(d_{vg}, d_{vb} : v_{vb})$ if $d_{vg} \in d_b \land d_g \in d_{vb}$ and $v_{vb}$ and $v_b$ have all but one bound dimension and value in common. The facet $f(week, zone : z_1, month : m_1)$ pivots on zone $z_1$ from the facet example, and is therefore its pivot facet.

**Explorability of the Cube:** It is clear that in our model, the user is 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. First, a group $g = d : v$, can be obtained from $|d|$ 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 each other 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 complete exploration of the cube lattice. Note that we do not require users to follow only the listed traversals – faceted traversals simply reduce the space of successive queries for speculation.

**Effectiveness of the Faceted model:** The four traversals mentioned above are both intuitive and sufficient to explore the entire data cube. The parent, child, and pivot traversals are inspired by roll-up, drill-down, and pivot operations respectively. It is possible to add
more traversal types, especially by mining a user’s query history for common “patterns” of analysis, e.g. keeping the bound dimensions the same and changing the group by dimension. Such extensions are easily pluggable into our system, but not required – the four traversals described above are intuitive and powerful enough to traverse the cube.

2.2.2 Distributed Execution

**Table Shards:** The interactive nature of our use case necessitates the approximation of results by executing queries over a subset of the data. We use *sharded tables* to achieve distributed and sampled execution of queries. A sharded table contains a subset of the rows of a SQL table and the concatenation of all shards across nodes represents the entire dataset. Each node may contain multiple shards. A sharded table is the atomic unit of data in our system: updates are performed at the granularity of the shard level, and each session makes the assumption that the list of shards and the shards themselves do not change. The coordinator maintains the catalog of the shards, which aids in the distribution of the workload. Upon an update or deletion of a shard, this information can be conveyed to the coordinator, enabling the handling of rapidly changing data. An in-memory LRU result cache is maintained at the workers. Previously executed queries can be answered using a simple lookup, reducing response times.

**Random Samples and Sharding:** Even though the data may fit within the memory of the same physical machine, it is sharded to enable a fine-grained control over sampling and caching. In single-threaded execution engines such as PostgreSQL, this enables execution over larger data sizes simultaneously, since different queries can run in parallel on different subsets of the data.
A random sample needs to satisfy two criteria. The first criterion states that every tuple should have the same probability of occurrence in the sample. The second criterion states that all possible samples of a given size should be equally likely and, thus, the inclusion of a tuple in the sample should not affect the inclusion probability of another [37]. *DICE* randomizes the tuple sequence before splitting the data horizontally into shards. The probability of a tuple being in a set of sequential tuples depends only on the probability of it being present at a given index, which is the same for every tuple due to randomization, thereby satisfying the first criterion. Again due to randomization, the location of a tuple is independent of the location of another, thereby, satisfying the second criterion. This results in every shard being a random sample. Further, union of shards results in a random sample without replacement since the union of random samples is a random sample and sharding results in the samples being disjoint. The principled way of sampling is computing separate samples for each query. However, this is expensive and can require a full table scan of the data. Thus, in order to avoid correlations due to groups of tuples or samples being used repeatedly, a common technique is to randomize the underlying tuple sequence and create new samples periodically [5].

### 2.2.3 Querying over Table Shards

A sample of the data is constructed online by choosing random table shards during run-time, allowing for random sampling. The union of shards selected for answering a query results in a random sample of the relation as explained above. We use standard sampling concept of stratified sampling [37] for estimating the error bounds of a query (Sections 2.3 and 3.3.1).
Given the preliminaries and definitions, in the naive case, the problem of ad-hoc cube exploration using the facet exploration model is simply that of successively executing each query received at a given sampling rate. We formulate our problem as the following:

For a facet session \( \vec{F} \), where each ad-hoc facet query \( f_i \) is expected to execute at a certain sampling rate, and the expected 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 \( \tau_V \), return \( f_i \) as quickly as possible to the end-user, preferably within the interactive threshold \( \tau_I \).

**Accuracy Gain Heuristic:** In order to schedule speculative queries at different sampling rates, we need to know the reduction in sampling error at different sampling rates. However, it cannot be known before actually sampling the data. Hence, we use a well-known property of standard SQL aggregates of the variance being approximately proportional to \( \frac{1}{n} \), where \( n \) is the number of tuples considered to estimate the improvement in accuracy. Thus, the standard deviation will be proportional to \( \frac{1}{\sqrt{n}} \). Therefore, we can estimate the future gain in accuracy based on the sampling rate. Thus, the estimated gain in the accuracy due to a unit sampling rate increase can be given as

\[
\text{AccuracyGain}(R_{curr}) = c \times \left( \frac{1}{\sqrt{R_{curr}}} - \frac{1}{\sqrt{R_{curr} + 1}} \right)
\]

(2.1)

where \( R_{curr} \) is the current sampling rate and \( c \) is the constant from the proportionality heuristic.

With more time permissible, we issue the same query on multiple shards on multiple nodes progressively giving us 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 cast this problem to fit the DICE framework in Section 2.4.

### 2.3 Variance Combination

We build upon stratified sampling [37] for our sampling framework. Error bounds for aggregation queries are based on the variance (across samples) of the measure for each cube group. As an initial step, we combine the variance for the same group across multiple shards, after which the variances across multiple groups can be combined to give an error estimate for the entire query.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s^2_h$</td>
<td>variance of the group $h$</td>
</tr>
<tr>
<td>$n_h$</td>
<td>number of tuples in the group $h$ in the sample</td>
</tr>
<tr>
<td>$n_{hi}$</td>
<td>number of tuples belonging to the group $h$ from the $i^{th}$ query</td>
</tr>
<tr>
<td>$n$</td>
<td>total number of tuples in the sample</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 the queries</td>
</tr>
<tr>
<td>$v_{hi}$</td>
<td>variance of the group $h$ from the $i^{th}$ query</td>
</tr>
<tr>
<td>$p$</td>
<td>proportion of tuples selected by the where clause</td>
</tr>
<tr>
<td>$V[\theta]$</td>
<td>variance of the estimator for the parameter $\theta$</td>
</tr>
<tr>
<td>$H$</td>
<td>number of groups in the union of all the queries</td>
</tr>
<tr>
<td>$N_h$</td>
<td>number of tuples in group $h$ in the dataset</td>
</tr>
<tr>
<td>$N$</td>
<td>number of tuples in the dataset</td>
</tr>
</tbody>
</table>

Table 2.1: List of notations used in this section.

**Combining variances within groups:** In order to deliver results at higher sampling rates, DICE runs the same query on multiple randomly chosen shards on multiple nodes. This results in the same cube group 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 \texttt{AVG}, \texttt{SUM}, and \texttt{COUNT} is straightforward, we use the following technique for combining variance

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

(2.2)

where \texttt{numQ} is the number of queries that a query needs to be replicated to. Thus, we get the requisite statistics for a combined group across all the replicated queries.

Continuing our motivating example, the faceted representation of the query is \( f(\text{rack}, \text{hour} : 6, \text{datacenter} : \text{EU}) \) with the measure \texttt{AVG} and measure dimension \texttt{iops}. We append the \texttt{COUNT} and \texttt{VARIANCE} measures to the queries since they are needed as given in Equation (2.2) to combine variances for the same group across multiple queries. Assume that the query is run on a single shard on 2 nodes and results into a sampling rate of 10%, returning us groups and the corresponding measures from the two queries respectively as:

\[
\begin{align*}
\{ & \{ \text{rack} : 1, \text{hour} : 6, \text{datacenter} : \text{EU}, \text{AVG} : 10, \text{COUNT} : 5, \text{VARIANCE} : 4 \}, \\
& \{ \text{rack} : 2, \text{hour} : 6, \text{datacenter} : \text{EU}, \text{AVG} : 12, \text{COUNT} : 6, \text{VARIANCE} : 2 \} \}, \\
& \{ \{ \text{rack} : 1, \text{hour} : 6, \text{datacenter} : \text{EU}, \text{AVG} : 5, \text{COUNT} : 8, \text{VARIANCE} : 1 \}, \\
& \{ \text{rack} : 2, \text{hour} : 6, \text{datacenter} : \text{EU}, \text{AVG} : 6, \text{COUNT} : 7, \text{VARIANCE} : 2 \} \}. 
\end{align*}
\]

Plugging in the values from above into (2.2), we get the variance for the combined group \([\text{rack} : 1, \text{hour} : 6, \text{datacenter} : \text{EU}]\) as \( s_1^2 = 8.32 \) and for \([\text{rack} : 2, \text{hour} : 6, \text{datacenter} : \text{EU}]\) as \( s_2^2 = 11.52 \).

**Combining variances across groups:** From the variances 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 algebraic measures \texttt{SUM}, \texttt{AVG}, and \texttt{COUNT}. From the standard sampling theory, the variance of the estimator for the measure \texttt{SUM} can be given as:

\[ \hat{V}[^\hat{\gamma}] = \sum_{h=1}^{H} N_h^2 \left( 1 - \frac{n_h}{N_h} \right) \frac{s_h^2}{n_h} \]  

(2.3)
The variance of the estimator for the measure $\text{AVG}$ can be obtained by dividing the above value by $N^2$.

For the measure $\text{COUNT}$, we can use the proportion estimator since the where clause acts as the indicator function and thus the variance of the estimator for $\text{COUNT}$ can be given as:

$$\hat{V}[^{\hat{p}}] = \left(1 - \frac{n}{N}\right) \frac{\hat{p}(1 - \hat{p})}{n - 1}$$ (2.4)

The above formulae cannot be used as they are since we cannot know the value of $N_h$ without accessing the entire data. We resolve this issue by estimating $\frac{N_h}{N}$ by $\frac{n_h}{n}$ and $\frac{n_h}{N_h}$ by the sampling rate.

Again plugging in the values we get, $\hat{y} = 6.92 \cdot 13/26 + 8.77 \cdot 13/26 = 7.85$ and $\hat{V}[^{\hat{p}}] = (\frac{13}{26})^2 \cdot (1 - 0.1) \cdot (8.32/13 + 11.52/13) = 0.35$

The error will be given by

$$\frac{\text{ConfidenceInterval}}{2 \cdot \text{Estimate}} = z_{\alpha} \frac{\sqrt{V(\hat{\theta})}}{\hat{\theta}}$$ (2.5)

where $\hat{\theta}$ is the estimate of the measure parameter and $V(\hat{\theta})$ is the variance of the estimate.

Thus, for a confidence of 95%, we can get the standard error for the query as $\frac{1.96 \cdot \sqrt{0.35}}{7.85} = 0.15$ (since $V(\hat{\theta}) = 0.35$, $\hat{\theta} = 7.85$, and $z = 1.96$ for 95% confidence level). This results in the accuracy being 85%.

2.4 The DICE System

2.4.1 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 result
for each query is inspected by the user for a small amount of time, after which the next query is issued. We consider this as a hidden opportunity – the database is simply waiting for the user to issue the next query. In light of this, our solution is to **utilize this waiting time to speculate, execute, and cache the most likely followup queries at the highest quality possible.**

While the concept of speculative execution is an intuitive one, there are several challenges to 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 followup queries? What is the strategy to employ to execute and cache likely queries? In a sampling approach, what is the sampling rate to run a speculative query at, given interactivity constraints? Finally, is there a singular framework to combine these problems into a cohesive, unified system?

Given these challenges, **DICE** solves the problems by using three complementary strategies. First, it performs **speculative query execution**, by caching results of likely followup queries, allowing for reduced latencies for ad-hoc query sessions. The enumeration of the likely followup queries is made possible by the **faceted** model of data cube exploration described in Section 2.2.1. Second, **DICE** employs a novel query execution architecture over a distributed database, executing queries piecemeal over individual table shards and then assembling them in a post-processing step. This architecture, in turn, allows for the **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.
2.4.2 System Architecture

Our architecture employs a hierarchical coordinator-worker approach, such that all queries are issued to the coordinator, and responded to by the coordinator. In line with the setting described in Section 2.2.2, each worker 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 worker nodes is maintained at the coordinator. The workers maintain an in-memory LRU cache for the results. In a fast-changing database, table shards can be atomically added and deleted from the workers, and the coordinator’s catalog can be updated, allowing for querying over rapidly changing data.
2.4.3 Query Flow

The high-level query flow of DICE is as follows: each ad-hoc query is rewritten and federated to the worker nodes, where it is executed. The results are returned, aggregated, and presented to the user. Next, the accuracy queries are run. Then, a set of speculative queries is executed till the next user query is received, with the goal of increasing the likelihood 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 its results are cached at the workers at a high sampling rate, thus reducing the latency of the overall ad-hoc query.

**User Query:** At startup, the coordinator makes sure that all workers are running and are ready to accept queries. On receiving an ad-hoc query, it is rewritten into multiple queries, one per required random table shard and passed to each worker. Since data is horizontally distributed across all worker 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 worker returns the results back to the coordinator, where the results are aggregated, and this information presented to the user.

**Accuracy Query:** The user query with the additional measure of \( \text{VARIANCE} \) is run after the results for the user query are returned. The variance for the same group across multiple queries is then aggregated and presented to the user.

**Speculative Queries:** Upon completion of the accuracy query, the coordinator 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 speculation 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 distributed amongst the workers in a round-robin fashion. Each worker issues, in an increasing order of rank, a
predefined number of concurrent queries to its database and populates the results in its cache (speculative query results are not sent to the coordinator). Upon receiving the next user query, the worker kills all currently running speculative queries.

**Successive User Query:** When the next ad-hoc query arrives, it is again rewritten and federated to the workers. If the exact query is cached, the result of the ad-hoc query is materialized from the cached result. If it is not cached it is executed on the database. The caching of speculated queries drastically reduces ad-hoc query latency and allows for a fluid and interactive data cube exploration experience.

### 2.4.4 Prioritizing Speculative Queries

As is clear from the query flow and the faceted model, each ad-hoc query can yield a significantly large number of speculative queries. Given the bounded time available for execution, it is typically not possible to execute all speculative queries. Thus, it is necessary to prioritize speculative query execution such that it maximizes the likelihood of results for the successive query being returned from the cache. This can, in turn, be done by maximizing the overall gain in accuracy, as discussed in Section 2.2.3. The selection of the maximal subset can be modeled as a linear integer programming problem as follows:

**MAXIMIZE:** \( \sum_{q \in Q} \text{Prob}(q) \cdot \text{AccuracyGain}(SR) \cdot x_q \)

**SUBJECT TO:** \( \sum_{q \in Q} \text{Time}(q) \cdot x_q \leq \text{totalSpecTime} \)

**WHERE:** \( x_q \in \{0, 1\} \)

Here, \( \text{Prob}(q) \) gives the probability of a query \( q \), which should be obtained from the query logs, \( Q \) is the set of all speculative queries at all sampling rates, \( \text{AccuracyGain}(SR) \) is the estimated gain in sampling accuracy, which depends on the sampling rate \( SR \) of \( q \) as
described in Section 2.2.3, Time(q) is the estimated running time, and totalSpecTime is the expected total speculative time.

Considering the input parameters, it is not possible to solve the above optimization problem in sub-second latency. We expect the majority of the query execution cost to be typically due to an in-memory table scan over identically sized data if the table shards are pre-loaded in the memory. It is not possible to load the entire dataset into memory but definitely a significant fraction which in our experiments was up to 20% such that the error bars for most of the groups were small. This lets us assume unit execution time for each query over a shard. In that case, 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. Therefore, the solution to the problem of choosing of the best queries that yield the highest overall accuracy gain turns into a greedy selection problem, as given below.

**Greedy Approach:** The greedy cost-based approach prioritizes the execution of the most likely queries that provide the highest overall accuracy gain. We represent the score of a query q at the sampling rate of SR as Prob(q) · AccuracyGain(SR).

In the case of multi-query optimizations such as unification (described in Section 2.4.6), where multiple queries are grouped together into a unified query $Q = q_{1..n}$, the score can be represented as $\sum_{q \in Q} Prob(q) \cdot AccuracyGain(q)$. 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 2.5.
In the case of sibling traversals for ordinal dimensions, a user is more likely to choose the changed bound dimension value closer to the current value. We use a heuristic that the distribution of the probability of the value that the changing dimension in the where predicate takes can be given as \( P(newVal) = O \left( \frac{1}{|newVal - oldVal|^2} \right) \). Let the set of speculative sibling queries and their probabilities be \( SQ = \{ S_Q1...n \} \) and \( P = \{ P_1...n \} \) respectively. We redistribute the sum of these probabilities between queries in \( SQ \) as \( P(x) = \frac{1}{c*(x-oldVal)^2} \), where \( c \) is the normalization constant given by \( \sum_{y \in \Upsilon} \frac{1}{(y-oldVal)^2} \) where \( \Upsilon \) is the domain of the changing dimension.

### 2.4.5 The DICE Algorithm

We are now able to illustrate both the overall model of the system (Algorithm 1) and the DICE algorithm (Algorithm 2) for selecting speculative queries.

```
EXPLORE(User u)
1  //CF : Current Facet
2  CF = null
3  while True
4    do
5      Query q ← TRAVERSE(u,CF)
6      Results r ← EXEC(q)
7      Q_spec ← ENUMERATE_SPEC(q)
8      P_spec ← DICEPLANDETERMINE(Q_spec,CF)
9    for each node n, queries Q in P_spec
10      do //parallel loop till next user query
11        NODE-EXEC-ALL(n,Q)
```

Algorithm 1: Core Exploration Loop

Algorithm 1 (Core Exploration Loop) describes the overall DICE cube exploration system. A user first selects a query (Line 5), which is then executed (Line 6). The system then enumerates all the different possible speculative queries based on the cube exploration
model described earlier (Line 7), and ranks them (Line 8). It then distributes the workload across all the available nodes (Lines 9–11). Next, in Algorithm 2, we formally describe how DICE ranks the speculative queries at different sampling rates.

Algorithm 2 (DICE Execution Strategy) starts by first finding out the normalized probabilities \( P_{\text{spec}} \) given a set of speculative queries (Line 1), and reweighting probabilities of the sibling queries as described in Section 2.4.4 (Line 2). Next, it generates the vector of the estimated accuracy gains for all sampling rates (Line 3), and performs unification over all speculative queries (Line 4) as given in Section 2.4.6. Finally, it ranks the unified queries at different sampling rates by the product of their probabilities and their corresponding sampling rate accuracy gains (Lines 5–10) and returns the sorted queries (Line 11).

## 2.4.6 Optimization: Query Unification

We now detail unification, a technique used to speed up query execution. The number of speculative queries as a result of faceted traversals can be extremely large. Further, taking replication of queries due to the usage of table shards results into consideration, the total number of speculative queries is equal to the product of the number of table shards and the
number of distinct speculative queries. Hence, it is not feasible to run all the speculative queries for most real-world datasets at high sampling rates within interactive time bounds.

We can observe that the generation of speculative queries leads to several queries that differ only by the value of a single bound dimension. Unifying multiple such queries into a lesser number of queries becomes essential since concurrently running all of them will congest the system. We use two unification techniques to minimize the number of queries. The first is to unify \texttt{WHERE} clauses on a column into a \texttt{GROUP BY} on the column, and the second is to split a dimension’s domain into ranges, and issuing range-based queries. The results of these unified queries can be post-processed to extract results for the user query.

**Groupby Based Unification:** Multiple queries can be unified into a single query by replacing the bound dimension that takes multiple values by a \texttt{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 the current user 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:** \( \{ \forall d_i : d_i \in \overrightarrow{d} - \overrightarrow{d_b} : f(d_g, d_i, \overrightarrow{d_b} : v_b) \} \)

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

**Range Based Unification:** At very high cardinalities, the problem of the very high number of speculative queries is not resolved by the groupby-based unification since the result set is expected to be large. Unifying the queries into *ranges* was found to be extremely useful. We convert multiple speculative queries \( f(d_g, d_b : v_1..v_n) \) into fewer range-based speculative queries \( f(d_g, d_b : [v_1..v_{n_1}]) \), \( f(d_g, d_b : [v_{n_1}..v_{n_2}]) \) .. \( f(d_g, d_b : [v_{n_{k-1}}..v_{n_k}]) \). The choice between range-based and groupby-based unification depends on the column cardinality and is a tunable parameter. This parameter can be obtained empirically using the marginal distribution of the column.

An interesting observation with range queries is that even with careful tuning of the ranges, the cardinality of the data for each range-unified query is large enough to motivate the use of an index on range-unified columns. Thus, we only index dimensions with very high cardinalities. While this introduces variability into our cost model, the lack of a good determiner for the cost of a range-unified query and aforementioned lack of a fast solution to our linear integer programming problem compels us to invoke Occam’s Razor and use a unit cost in this case and the resultant greedy algorithm for query selection.

### 2.4.7 Optimality of DICE

As described in Sections 2.4.4 and 2.4.6, we cast the linear integer programming problem of maximizing the overall accuracy under the constraint of maximum allocated time into a greedy algorithm of choosing the new query at an additional unit sampling rate. As we
will not know the accuracy gain without actually running the query, approximating the gain using the estimated accuracy gain is a sound assumption to make. Without indexes and joins, all queries will take similar times to finish. Therefore, 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.

2.5 Experiments & Evaluation

2.5.1 Experimental Setup

DICE is implemented in Java running on Sun Java 6 VMs and uses PostgreSQL 9.1 as the database for each worker node. By default, we discard the first run of each experiment and report the average of the following three runs (runs were nearly identical for all experiments, with no outliers, also observed by the low standard deviation). We perform an exhaustive analysis of the DICE system over a variety of cluster configurations, workloads, and algorithms for our metrics, as described below.

Cluster Configurations: CLUSTERSMALL is a private cluster built on commodity hardware with only DICE running during the experiments. The coordinator node has 1 Quad Core 3.30GHz Intel i5 CPU, 16GB DDR3 RAM @1333MHz, and 256GB SATA HDD. The 15 worker nodes each possess 1 Quad Core 2.13GHz Intel Xeon CPU, 4GB DDR2 RAM @667MHz, and 720GB SATA HDD. Nodes are connected over a Gigabit Ethernet switch. Each worker contains 4 workers. CLUSTERCLOUD is an Amazon EC2 configuration of 1 coordinator and 50 workers of the cl.xlarge type, each with 7GB Memory and 8 Virtual cores, powering 8 workers per worker node. All nodes for both configurations run Ubuntu Linux 12.04 LTS.
**Dataset:** Our generated dataset conforms to the example schema provided in Section 2.2.1, and comprises 1 billion rows sharded uniformly across all nodes with a default table shard size of 1M rows. The distributions and cardinalities are:

- **location**: uniform:zone{10}:datacenter{100}:rack{1000},
- **time**: gaussian:month{12}:week{52}:hour{24},
- **iops**: zipfian:{10000}.

Each table shard is 102MB on disk, with a data size of 81MB and index size of 21MB, yielding 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.

**Workloads:** The user was asked to explore the dataset taking into consideration the faceted exploration model using a popular BI tool. Query logs from the tool were used to derive the workload. A workload depicts a user query session of 10 facet traversals, with the measure function \( \text{AVG} \). Unless stated, 3 workloads were used for each experiment and with the aforementioned 3 runs, results in a sample size of 90 queries. The viewing latency threshold \( \tau_v \) is fixed at 5000ms.

**Algorithms:** We compare five different algorithms: ALGO\text{NOSPEC} 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. ALGO\text{RANDOM} represents distributed querying using query speculation, but the queries chosen to be speculated are selected randomly from the set of possible facet traversals. ALGO\text{UNIFORM} selects speculative queries uniformly from each type of facet traversal. ALGO\text{DICE} uses the
DICE speculative query selection technique. 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.

**Metrics:** AVERAGE LATENCY is measured in milliseconds as the average latency of a query across sessions and runs. We also depict ±1 standard deviation of latency using error bars in most of our results. AVERAGE ACCURACY is measured as the absolute percentage deviation of the sampled results from the results over the entire dataset.

2.5.2 Results

2.5.2.1 Impact of Data Size

We observe, in Figures 2.3 & 2.4, 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 almost linearly, and exceeds 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 stays 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. A 1-tailed t-test confirms (p-value 0.05, t-statistic 58.41 > required critical value 1.662) that ALGODICE’s speedup over ALGONOSPEC is statistically significant. Another observation is a performance envelope with ALGOPERFECT exists – there are several constant-time overheads which could be further optimized, an
opportunity for future work. Figure 2.4 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.

![Figure 2.3: Varying Size of Dataset (CLUSTERSMALL)](image)

### 2.5.2.2 Sampling & 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 data skew. As described in the schema, our dataset contains a multitude of distributions across all dimensions. Second, the selectivity of queries in the workload will impact the sensitivity of error. Third, the placement of the data is a significant
Figure 2.4: Varying Size of Dataset (CLUSTER CLOUD)

Figure 2.5: Accuracy over a workload

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 2.5, 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.
2.5.2.3 Number of Worker Nodes

We vary the number of worker nodes in Figure 2.6, while keeping the size of the data constant at 200M rows. As expected, for all algorithms, latencies decrease as the number of nodes increases. An interesting observation is made for ALGODICE – 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 or caching) or ALGOPERFECT (exactly one ad-hoc query being cached).

2.5.2.4 Cache Hit Variability

Since the cache hit rate is a key contributor to the average latency of a session, in Figure 2.7 we study how the cache hit rate varies with the sampling rate for a fixed cache size. We use the cache hit rate as a proportional measure of the prediction quality. Higher cache hits are a direct result of high quality of speculation. We achieve close to a 100% hit rate for 50 million sampled rows. As we increase the sampling rate, we see the cache hit rate
Figure 2.7: Cache Hit Change with Sampling Rate Change

decreasing nearly linearly since the total number of speculative queries increases linearly
with the sampling rate.

2.5.2.5 Sample Session

As an example, we present in Figure 2.8 the trace of a single cube exploration session
for ALGONOSPEC, ALOGDICE, and ALOGPERFECT 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 for the reader, the session for each algorithm is executed separately.) The Y-axis represents \textsc{AVERAGE LATENCY}. Cache hit rate for \textsc{AlgOdice} is shown as a label above the bars. The cache hit rate for the first query is 0.0, since there has been no speculation and the caches are empty. \textsc{AlgOdice} performs almost as well as \textsc{Algoperfect} with hit rates equal or closer to 1.0.

\subsection*{2.5.3 Real-world Usage and User Study}

\textbf{Real-world Query Logs:} To evaluate the real-world efficacy of the facet model, we procured a real-world query log of ad-hoc analytical queries by real users on a production system generated HIVE data warehouse of an Internet advertising company. Considering only the aggregation queries (with the \texttt{GROUP BY} clause), the log spanned 509 queries. Amongst them 46 query sessions were detected, which comprised of 116 queries i.e. 22.97\% of the queries. The traversals described in the \textsc{Dice} model were found to cover 100\% of the session-based queries, demonstrating that our traversal model is indeed expressive enough to allow for significant speedups (the remainder are executed traditionally, without speculation).

\textbf{User Studies:} We performed a user study to compare the effectiveness \textsc{dice} over traditional methods. The study was performed with 10 graduate students across the department who were knowledgeable in databases and data cubing, determined using a pre-test. The users were given a pre-task tutorial on data cubing and our data model. They were asked to explore the cube using the faceted model for 10 ad-hoc queries of their choice. They were not told if the \textsc{dice} speculation was turned on or off (50\% of the users each). After the session, the user’s query session was repeated in the other mode (speculation was turned on if it was off before and vice-versa) to get comparable times. Care was taken to avoid different biases.
The pre-task tutorial avoided bias against the prior knowledge of data cubing and our data model. Having only 1 total task that lasted less than 10 minutes prevented fatigue bias, and the same user workload being re-run (automated) for the alternate mode avoided learning effects. The test algorithm being split equally dealt with carryover effects.

The mean and standard deviation for *Time-To-Task* for the entire query session of 10 queries using ALGODICE were 47757 ms and 937 ms and for ALGONOSPEC were 54506 ms and 3111 ms: i.e. on average users queried 7 seconds faster with ALGODICE. Consider null hypothesis as ALGODICE execution time to be no different than ALGONOSPEC time and alternate hypothesis to be that ALGODICE is faster than ALGONOSPEC. The query session time which consists of query execution time, query input time, and result view time is significantly lesser for our method, ALGODICE, compared with ALGONOSPEC based on a 1-tailed t-test (t-statistic value of 21.77 > 1.833 needed for a p-value of 0.05).

**Speculation Noticeability:** While DICE speedups are objectively significant, an important question for a user-involved system is: Can users notice the difference and have a preference? To test this, the users were asked to report which query session (i.e. with or without DICE) they found out to be faster. Results were unanimous: All our users preferred the ALGODICE session over ALGONOSPEC. Clearly, reduction in query times due to usage of ALGODICE speculation is indeed noticeable to the user.

**User Satisfaction:** At the end of the query session, users were asked to rate their satisfaction (10: extremely happy, 5: neutral, 1: extremely unhappy) for both the faceted traversal model and the overall system. For the **traversal model**, the main criteria they were asked to take into consideration were the traversals allowed under DICE, any extra traversals they thought it lacked, and the ease of traversal. The mean rating for the faceted model was 7.9 with a standard deviation of 1.54. Consider the null hypothesis of the faceted model ratings
being equal to 5 (i.e. neutral) and the alternate hypothesis of the faceted model rating being greater than 5. The value of the t-statistic was found out to be 5.67, which is higher than the critical value of 1.833 needed for a 1-tailed t-test for a p-value of 0.05, showing that the traversal model satisfaction was statistically significantly better than random/neutral response. Additionally, for the overall DICE system, the average User-Satisfaction was very high: 8.7 with a standard deviation of 0.82, summarizing our overall assertion that DICE not only provides objective speedups, it also provides a significantly better experience for the end user.

2.6 Related Work

Cube Exploration: While the original cube paper [57] provides for a variety of operators, facet traversals introduced in this work are typical of interactions on analytics user interfaces. Work by Sarawagi et al. on mining of interesting regions [128] and exploration operators [129] can be easily plugged into our speculation framework. Kamber et al. [82] have discussed metarule exploration, and dynamic exploration on cube subsets have been discussed in [89]. Our contribution is towards improving interactive exploration in a session context.

Cube Materialization: Materialization strategies range from full-cube materialization over MapReduce [108] to region-specific materialization [33] to selective partial materialization. Optimization techniques exist for optimizing intra-query parallelization [7], but do not consider multiple queries as part of an interactive session.

Distributed Query Execution: Ad-hoc analysis over large datasets has been made popular with the availability of declarative query languages such as SCOPE [25], Pig [115], and Hive [143], which translate to MapReduce-oriented flows, and is not ideal for interactive
workloads. Ideas such as columnar storage layouts [21,62,139], hierarchical execution [102], distributed database hybrids [1], and main-memory engines [51] have achieved low latencies when querying over large datasets, resulting in a spurt of development activity in this area, resulting in implementations such as Drill, Impala, Tez, PivotalHD, HAWQ, Peregrine, and Druid, projects that target single query execution latency.

**Prefetching:** The idea of speculative execution of queries and prefetching results has been discussed before [135]. PROMISE [127] investigates the likelihood of future queries and can be used to supplant the workload-based approach in our work. Improvements in speculation quality based on ideas in these papers can be used to better prioritize and sample our speculative queries.

**Online Aggregation:** In addition to the initial work in online aggregation by Hellerstein et al. [68, 69], applications to continuous, parallel, and distributed contexts have been studied [14,38,119,154]. Our system builds upon these ideas in a distributed cubing environment, combining user-directed techniques of speculative execution and sampling.

**Sampling-based Estimation:** There is significant prior work in using sampling for approximating query results [112]. Jin et al. [75] detail the approximation of OLAP queries using pre-summarized statistics. Wang et al. discuss [152] data placement, [156] details the computation of errors for a GROUP BY query over multi-table joins, and [92] discuss a sampling-based framework to materialize cubes. BlinkDB [4] performs an offline sampling step of multiple column combinations. As mentioned before, the ideas presented in BlinkDB are orthogonal to both the faceted exploration model proposed by our work and the speculation-based execution architecture. Strategies for stratification using prior workloads [30] and methods to increase sensitivity for low-selectivity attributes [150] have also been considered before.
2.7 Conclusion & Future Work

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 performance 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, this can be considered when prioritizing facets to speculatively execute. DICE can be trivially extended by modeling user behavior through query logs and plugging in new traversal types. Another possible extension is to combine methods for offline and online materialization of data cubes by identifying the fraction of the cube to fully pre-materialize. A possible approach is to materialize an approximate and compressed representation [49] of the data cube, and use the online execution step to increase the quality of the answer based on the approximate model [65].
Chapter 3: A Session-Based Approach to Fast-But-Approximate Interactive Data Cube Exploration

3.1 Introduction

Approximate querying helps avoid computation over the entire dataset by accessing a sample of the data and aims to provide results within error bounds that the user is satisfied with. We have seen in DICE that by running a query over a sample of the data, it is possible to drastically reduce the execution time while returning an approximate result with error estimates to the user. It is important to note that the results obtained using a sample of the data have little value if not provided with the corresponding error bars as well, which inform the user about the accuracy of the results. Errors for commonly used measures, e.g., SUM, MEAN, are typically evaluated using the variance of the sample.

Reusing Variance Computation: While variance is an essential component of error estimation, computing it is expensive (we theoretically demonstrate that this is a critical concern in Section 3.3.2 and show the faster solution empirically in Section 3.5.3.6). Hence, in order to leverage the benefits of approximate querying more effectively, it is essential that we expedite variance computation. An intuitive way to do so is to reuse the variance calculations. Sesame is the first to focus on this critical aspect – in DICE, queries with variance are run after the user query, burdening the system with an additional query, and
Figure 3.1: Common Error Visualizers: The figure depicts a sampled aggregation query with some commonly used representations to visualize errors. It shows Error estimates (often represented visually as error bars), which are an important part of the output of sampled aggregation queries. ERR() represents error estimates such as standard deviation, standard error, or other variance-related computations that are expensive to compute.

significantly delaying knowledge of the error estimate to the user, while in BlinkDB, the authors use error profile to approximate the sample variance. In contrast, our system reuses variance computation in addition to the results. In this work, we provide new strategies for cache and result reuse by leveraging previously unexplored properties of statistical measures such as variance and standard error to answer sample-based queries.

There are three key challenges when building an architecture for session-oriented sampled aggregation. Our first challenge and key design goal is, how can we provide the exact and fast computation of variance within interactive response times? Secondly, given the opportunities provided by session-based workloads, how can we find out the best queries to speculatively execute taking into consideration all the queries issued so far in the query session? Third, in terms of result reuse, given a choice between different parts of the cache that can be used to answer a query, which ones should be chosen? As shown in our experimental evaluation, Sesame provides lower query response times and consistent performance both within a session as well as between query sessions, thus enabling a smoother interactive user experience.
3.2 The Sesame Model

In this section, we describe the Data, Query, and Session Models of Sesame. We use the standard data cube model since it is the ideal and natural model for aggregate querying and fits well in Sesame’s paradigm. Sesame reuses result and error computations from previous queries in the session and pre-populates the session cache with likely follow-up queries (Figure 3.2).

3.2.1 Query Model

As described in Section 3.1, Sesame uses a session-based query model due to queries occurring in sessions in OLAP scenarios. Similar to DICE, Sesame uses the faceted cube exploration model.
Sesame uses the traversal operators used in DICE [77], namely Parent (similar to ROLLUP), Child (similar to DRILLDOWN), Sibling (changing WHERE predicates), and Pivot. These operators were modified to allow for multiple GROUP BY dimensions.

We also introduce a new operator called RESAMPLE. It is used when the user is not satisfied with the results obtained at the current sampling rate and wishes to obtain the results at a higher rate. This can occur when the error rates are higher than what the user is happy with. RESAMPLE then runs the user query at the additional rate and combines the results with the already computed results.

3.2.2 Session Model

In a query session, it is likely that the user forgets some of the previously seen results over time. Building upon the well-established body of work in cognitive science, we use the forgetting curve [12] to model the memory retention of a user over a query session. The forgetting curve models the retention of a fact by \( e^{-c \cdot \Delta t} \), where \( c \) is a constant based on memory strength (empirically set at 0.15), and \( \Delta t \) is the time elapsed since observing the fact. In the case of multiple events, we define the potential for gaining information about the query as \( InfoGainPotential(q) \), and model it by \( 1 - \sum_{\Delta t \in T} e^{-c \cdot \Delta t} \), for the query \( q \), given different times \( \Delta t \in T \) since the results were last seen. We use these ideas to determine the set of speculative queries (Section 3.4.1).

3.3 Error Computation in Sesame

As mentioned in Section 3.1, variance is a popular and often necessary part of sampled aggregation queries, and can be more expensive to compute than measures such as SUM, MEAN, COUNT, etc. In a pre-experiment on a standard database system using a TPC-DS derived dataset and a real-user workload, we observed that including variance calculations
increases query response time by nearly 54.6%, in contrast with simply including the other measures given above. Thus, including variance can be substantially more expensive and can reduce the query response time benefits of sampled aggregations.

A common approach to expediting cube materialization and online cube computation is to distribute/parallelize the effort [107, 109], reusing prior computations for efficiency. In Sesame, we build on this concept in the following ways. Beyond just results, we use the fact that variance is, in fact, an algebraic measure, and that results from variance computation can be reused (Section 3.3.1). We compare various techniques that can be used to calculate variance, and detail which ones are viable for reusing results and which ones yield faster execution times (Section 3.3.2).

3.3.1 Error Reuse

There has been extensive work in reusing results for measures such as \textit{SUM}, \textit{MEAN}, etc. [34, 61]. However, Sesame is the first system that provides a framework for reusing the errors as well, which as elaborated in Section 3.1, is an important problem in the context of sampling-based aggregation.

A common representation of variance calculation is:

\[
v = \frac{1}{n-1} \left( |G| \sum_{i=1}^{\mid G \mid} n_i (m_i - m)^2 + \sum_{i=1}^{\mid G \mid} (n_i - 1)v_i \right)
\]

(3.1)

which we denote as \textit{Variance\textsubscript{Common}}, where \(v\) is the combined variance of all groups, \(n\) is the number of tuples in all groups, \(m\) is the mean of the entire sample, \(v_i\) is the variance of group \(i\), \(n_i\) is the number of tuples in group \(i\), \(m_i\) is the mean of group \(i\), and \(|G|\) is the number of groups.

We note that this representation of variance lets it be presented as an algebraic measure, as a function of mean, count, and variance of individual groups. In this fashion, we can
obtain the overall variance by combining the variance from the corresponding groups without having to touch the underlying data in the variance combination step.

Another representation of variance calculation is [153]:

\[ v = \frac{1}{(n)(n-1)} \left( n \sum_{i=1}^{n} x_i^2 - \left( \sum_{i=1}^{n} x_i \right)^2 \right) \quad (3.2) \]

which we denote as Variance\(_{Fast}\). Here, \( x_i \) items are present in a sample of size \( n \). We can see that by keeping a counter for \( \sum_{i=1}^{n} x_i \) and \( \sum_{i=1}^{n} x_i^2 \), variance can be computed easily and, thus, this representation of variance is also algebraic.

It is worth noting the differences between Variance\(_{Common}\) and Variance\(_{Fast}\) from the perspective of interactive query performance. In Variance\(_{Common}\), the values for the mean, count, and variance for each group need to be stored whereas in Variance\(_{Fast}\), we only need to keep track of \( \sum_{i=1}^{n} x_i \) and \( \sum_{i=1}^{n} x_i^2 \). Our experiments show that as expected the second technique is much faster (Section 3.5.3.6). We further elaborate on the reasons behind this in Section 3.3.2.

Another famous representation of variance developed by Welford et al. [153] computes the running values for mean and variance as follows:

\[ \mu_n = \mu_{n-1} + \frac{x_n - \mu_{n-1}}{n} \quad (3.3) \]

\[ v_n = v_{n-1} + (x_n - \mu_{n-1}) \cdot (x_n - \mu_n) \quad (3.4) \]

where \( \mu_n \) is the mean of \( n \) items, \( \mu_{n-1} \) is the mean of the first \( n-1 \) items, \( x_n \) is the \( n^{th} \) item, \( v_n \) is the variance of \( n \) items, and \( v_{n-1} \) is the variance of the first \( n-1 \) items. We denote this representation as Variance\(_{Incremental}\). We can clearly see that since it incorporates a single value at every iteration, it is unsuitable for larger scale reuse.
Another variance calculation formula derives from Equations 3.3 and 3.4 and presents a method for combining means and variances of 2 groups [27]:

$$\mu = \mu_1 + n_2 \cdot \frac{\mu_2 - \mu_1}{n_1 + n_2}$$

(3.5)

$$v_n = v_{n_1} + v_{n_2} + n_1 \cdot n_2 \cdot \frac{(\mu_2 - \mu_1)^2}{n_1 + n_2}$$

(3.6)

which we denote as $Variance_{BiCombination}$. However, this representation is not amenable to be calculated using a SQL query since it only combines two groups at a time. Thus, the only two representations that can be used in variance reuse are those given by $Variance_{Common}$ and $Variance_{Fast}$.

### 3.3.2 Efficiency of Variance Calculations

One of the reasons for $Variance_{Fast}$ to give results much faster than $Variance_{Common}$ is that it does not involve the calculation of the variance until the final step and only needs to keep track of $\sum_{i=1}^{n} x_i$ and $\sum_{i=1}^{n} x_i^2$. $Variance_{Common}$, on the other hand, needs to keep track of the variance, mean, and count at every step for every group. Another reason is the lack of support for nested aggregation queries in standard databases such as PostgreSQL, MySQL, etc. which further degrades its performance. Hence, from a perspective of interactive approximate query processing, usage of $Variance_{Fast}$ is preferable over $Variance_{Common}$.

### 3.3.3 Extensibility to Other Measures

So far we have seen how result reuse can be extended to variance. As variance is algebraic, standard deviation, standard error, and coefficient of variation can be termed as algebraic as well. Thus, these measures can also be reused.

The error reuse and caching framework of Sesame is not limited to simply these measures. It can also be extended to any user-defined measure whose variance can be expressed in
closed form as a function of the variance of one of the measure dimensions. For example, for a user-defined measure given by $a \ast \text{AVG}(\text{Agg}_1) + b$, where $a$ and $b$ are constants and $\text{Agg}_1$ is a measure dimension, the variance of the measure can be given in closed form as $a^2 \ast \text{VARIANCE}(\text{Agg}_1)$.

3.4 The Sesame Framework

*Sesame* presents a standard query execution workflow to the user while using a set of several complementary techniques. Figure 3.3 presents an architectural outline of the system. Once a user query is received from the interactive query session, it is executed and the results are returned. After finishing the execution of the user query, *Sesame* calls the *speculation* module to enumerate different follow-up queries and executes an optimal subset of them. When possible, results and error computations are reused from the novel lattice-based session cache, which is designed for cube-based querying. The session is highly interactive, targeting a near-sub-second response time per query. This necessitates an in-memory-only architecture, which *Sesame* employs. *Sesame* connects to a SQL-capable database as its backend to execute the queries, and uses query rewriting and post-aggregation to coordinate reuse. *Sesame* uses data sharding, which is a pervasive and pragmatic technique for large data platforms. While *DICE* performs some of the computation itself, *Sesame* is a pure query rewriting-based system. This is one of the reasons behind it giving large speedups.

We provide methods to generate the optimal set of speculative queries (Section 3.4.1). We show how the query cache can be modeled as a lattice and the benefits of doing so (Section 3.4.2). We provide techniques for determining which cached view should be used to retrieve results for a user query amongst different possibly reusable views to minimize the response time (Section 3.4.3 and Section 3.4.4). We further improve the
lattice-cache by providing techniques to assign shards amongst the different nodes of the lattice to maximize the probability of obtaining reusable results that belong to different shards, and consequently selecting the shards on which to run a user query (Section 3.4.5).

### 3.4.1 Session-Aware Speculation Schedule

As stated in Section 3.2.2, over a query session, a user is likely to retain only partial information about the results seen so far. Hence, *Sesame* uses the entire query session of the user to determine an optimal set of queries to speculatively execute and cache. Similar to *DICE*, *Sesame* also uses accuracy gain.

Hence, we model the selection of a subset of speculative queries to execute, given the queries executed in the session so far, as the following *Maximum Integer Linear Programming* problem.

**MAXIMIZE:** \( \sum_{q \in Q} \text{ExpectedGain}(q, SR) \cdot \text{InfoGainPotential}(q) \cdot x_q \)

**CONSTRAINT:** \( \sum_{q \in Q} \text{ExecutionTime}(q) \cdot x_q \leq \text{SpeculationTime} \)

**GIVEN:** \( x_q \in \{0, 1\} \)
where \( q \) is a speculative query amongst the set of all speculative queries \( Q \) (all possible followup queries according to the traversal model given in Section 2.2.1), \( \text{ExpectedGain}(q, SR) \) is the product of the probability of \( q \) being the next user query which is obtained from the query history and the estimated accuracy gain of running a query at the sampling rate of \( SR \), \( \text{ExecutionTime}(q) \) is the estimated time taken to execute query \( q \), \( \text{SpeculationTime} \) is the time we expect will be available to execute the speculative queries, and \( x_q \) indicates whether the query \( q \) is part of the selected queries.

The \( \text{SpeculationTime} \) is estimated based on the user behavior in the query session thus far. The optimal set of speculative queries is determined using the Gurobi Optimization Solver (www.gurobi.com). If the user interrupts and issues the next query while the speculative queries are being run, we stop the execution of the speculative queries and run the user query instead. If after running the chosen speculative queries, the user still has not run his next query, we choose amongst the rest of the speculative queries.

### 3.4.2 Lattice-based Session Cache

*Sesame* uses a novel yet intuitive cache structure based on materialized views in the backend database. We model the cache as a lattice mimicking the data cube model, with each region of the cube having a cache associated with it (Figure 3.2). This gives us a finer grained control and greater flexibility over the caching resource. It also helps us direct the caching resources better, as the caches near the current user region can be dynamically made larger than those further away using our session-based query model. The lattice structure of the cache also helps speed up the lookup for potential views that can be reused to answer the user query by reducing the size of the cache that needs to be searched, as described in
Section 3.4.4. The lattice-based session cache is also the basis of the shard partitioning strategy described in Section 3.4.5.

### 3.4.3 Optimal View to Reuse

For distributive and algebraic measures, it is clear that the results for a query can be obtained in its entirety by reusing the results of any one of its child queries. We discuss different scenarios that provide possibilities for result reuse and suggest the best decision in each case. Due to lack of indexes and memory-resident data, we can safely assume that the optimal decision is to choose the view that has the least number of tuples [65]. To the best of our knowledge, this is the first piece of work to enumerate the different options and provide the optimal solution in each case. We also present an extension in the case that the data is horizontally partitioned into shards. Similar to [65], we denote by $Q_1 \preceq Q_2$, if $Q_1$ can be answered in its entirety by $Q_2$.

**Rule 1:** Suppose $Q_1 \preceq Q_2 \preceq Q_3$ and we have to choose one amongst $Q_2$ and $Q_3$ to answer $Q_1$. In this case, $Q_2$ will have lesser number of tuples than $Q_3$, and should be preferred.

**Rule 2:** Suppose $Q_1 \preceq Q_2$, $Q_1 \preceq Q_3$, $Q_2 \not\preceq Q_3$ and $Q_3 \not\preceq Q_2$. In this case, there is no clear choice between using $Q_2$ or $Q_3$ to answer $Q_1$ based on their relationship alone. In this case, we choose the view with lesser number of tuples.

**Rule 3 (Sharding effect):** Consider the case where two views were the result of the same query running on two different shards and we had to choose between the two views to answer the user query. In this case, we again choose the view with the fewer number of tuples. In the case of data skew, this is an important case to consider.
3.4.4 Lookup Algorithm

The principles given above are used in designing the view lookup algorithm, where we choose the views used to answer a user query. In order to satisfy a user query, we look up the views in the cache that can completely answer the user query. The lattice structure of the cache helps this process since only views in the region that the query belongs to, or its descendant regions can be used to answer a given query. Thus, we can avoid searching through a large section of the query cache. The lookup algorithm for potential superset views can then be given as follows.

1. A candidate list of views that can be used to answer the user query is kept sorted by view size. Given a query for which we wish to lookup views that can be used to answer it, we look up first in the cache of the region that query lies in and add the usable views to the list.
2. Whenever all the views belonging to a region are used to answer the user query, we add the usable views from its child regions to the candidate list (Scenario 2).
3. Step 2 is recursed till the desired sampling rate is met or all the caches of the descendant regions of the user query have been searched.
4. If the sampling rate is not met, we run the user query on the shards that have not been used so far.

3.4.5 Shard Assignment

As mentioned before, Sesame uses data sharding where tuples in the dataset are randomized and divided horizontally into shards. It is clear that any selection of tuples from the randomized sequence including a subset of contiguous tuples and union of different subsets can be considered as a random sample without replacement of the underlying data (Section 2.2.2).
Using multiple views derived from the same base shard to answer a user query does not provide more information than a single one. Hence, it is preferable for the reusable views to have disparate shard origins. As a result, we need to choose shards that a query executes on so that it increases the probability of reusable views having distinct shard origins. We do so using a novel shard assignment strategy.

A query can be answered completely by using the results belonging to its region or its descendant regions. Keeping this in mind, we now describe the motivating example behind the shard assignment strategy. Consider a cube having 3 dimensions – $a, b,$ and $c$. The results from the regions $\{a, b\}$ and $\{a, c\}$ can be used to answer queries on region $\{a\}$. Having results available from $\{a, b\}$ and $\{a, c\}$ belonging to the same shard to answer a query on region $\{a\}$ will be a waste of precious reusable resource. Hence, the shards assigned to region $\{a\}$ should be distributed amongst regions $\{a, b\}$ and $\{a, c\}$ so that the intersection set of the shards belonging to $\{a, b\}$ and $\{a, c\}$ is null. Note that this is only a logical assignment. We base our shard assignment algorithm on this intuition. The algorithm can then be given as follows:

1. Assign weights to all the regions in the cube based on their relative frequency of occurrence in the query logs.
2. Assign all the shards to the topmost region of the cube.
3. Recursively divide the shards amongst its child regions based on their relative weights.

In order to take advantage of this shard assignment strategy, we present below its natural counterpart for selecting shards to run a user query on.

1. Given a query, the reusable views, and their corresponding shards, and the additional required sampling rate, search amongst the shards assigned to the current region to check if any shards are available from whom the results have not been already reused.
2. Recursively follow this strategy through its ancestors.

### 3.4.6 The SESAME Algorithm

In this section, we summarize all the design decisions involved in building Sesame and present the overall algorithm succinctly.

```plaintext
SESAME-SESSION()
1 ASSIGNEDSHARDS(AllShards, LatticeRoot)
2 while UserQuery
3 do
4 Views ← FINDREUSABLEVIEWS(UserQuery)
5 RUNQUERIES(UserQuery, Views)
6 if COUNT(Views) < SamplingRate
7 then
8 CountOfExtraShardsNeeded ← SamplingRate − COUNT(Views)
9 Shards ← FINDUNUSEDSHARDS()
10 RUNQUERY(UserQuery, Shards)
11 SpecQueries ← FIndOPTIMALSET(UserQuery)
12 RUNQUERIES(SpecQueries)
```

Algorithm 3: Overall Algorithm

Algorithm 3 describes the overall flow of Sesame. Initially, the shards are assigned to the different regions of the lattice (Line 1). Next, the reusable views are found (Line 4) and the user query is run on these views (Line 5). Based on the number of extra shards needed to meet the sampling requirement (Lines 6-8), we find the shards on which the query needs to be run (Line 9) and run them (Line 10). The optimal set of speculative queries are then found (Line 11) and run until the user issues his next query (Line 12).

Algorithm 4 formalizes the shard assignment strategy presented in Section 3.4.5. All shards are assigned first to the given region (Line 1). The shards are then distributed amongst the children regions based on their importance using the query logs (Line 2) and this procedure is recursively followed (Lines 3 – 5).
**Algorithm 4: Shard Assignment Algorithm**

```plaintext
ASSIGNSHARDS(AllShards, Region)
1    ASSIGNSHARDSTOSELF(AllShards, Region)
2    ASSIGNSHARDSTOCHILDREN(AllShards, Children)
3    for each Child in Children
do
4        ASSIGNSHARDS(ChildShards, Child)
```

**Algorithm 5: Finding Reusable Views**

```plaintext
FINDREUSABLEVIEWS(Query)
1    Views ← FINDEXACTMATCHINCACHE(Query)
2    // Find optimal views in region if sampling rate has not been satisfied
3    Views ← Views ∪ FINDOPTIMALVIEWS(Region)
4    // Find optimal views recursively in child regions if sampling rate still has not been satisfied
5    Views ← Views ∪ FINDOPTIMALVIEWS(Descendants)
6    return Views
```

**Algorithm 6: Finding Unused Shards**

```plaintext
FINDUNUSEDSHARDS()
1    UsedShards ← FINDSHARDNUMBERS(Views)
2    // Methods below use information about used shards and extra shards needed
3    UnusedShards ← FINDUSABLESHARDS(UsedShards)
4    UnusedShards ← UnusedShards ∪ FINDUSABLESHARDSFROMANCESTORS(UsedShards)
5    return UnusedShards
```

Next, we look at finding the shards to run a query on (Algorithm 6). First, the list of shards on which views have been computed is found (Line 1). Then, amongst the shards assigned to the region, we find the shards not present in the list of shards computed above.
Following that, we traverse up the lattice recursively choosing amongst the shards assigned to ancestor regions (Line 4).

### 3.5 Experiments & Evaluation

#### 3.5.1 Experimental Setup

*Sesame* is implemented in Java 7 and uses PostgreSQL 9.3 as the default database. It runs on an Ubuntu Linux 14.04.1 LTS system with a 24-core 2.4 GHz Intel Xeon CPU, 256 GB DDR3 @ 1866 MHz memory, and a 500 GB @ 7200 RPM disk. Experiments were carried out in an in-memory-only setting. Buffers and caches at all levels were flushed / purged before each experiment, and to avoid external / warmup effects, results from the first of the 4 iterations were discarded, and the mean and the standard deviation across the remaining 3 iterations are provided (as described in Section 3.5.4, error bars may not be visible due to uniformity in execution times across iterations).

**Dataset and Query Workloads:** Both data and query workload were derived from TPC-DS [117] since it is designed for evaluating the performance of data warehousing queries. TPC-DS contains four iterative query sessions that we use to design a single query session, *Workload_{TPCDS}*, consisting of 35 queries. We also collected a *real-user workload* from a user study of five graduate students who were asked to analyze the TPC-DS dataset using the industry-standard *Tableau* data analytics tool ([http://tableau.com](http://tableau.com)). Query logs from this usage were used to generate another query session (*Workload_{Real}*), comprising 21 queries. Unless otherwise specified, the experiments were performed using *Workload_{TPCDS}*. *Sesame* achieved similar performance over smaller sessions. The data of size 20.7 GB was partitioned horizontally into shards, with each shard having 100,000 tuples, resulting in a total of 768 shards.
3.5.2 Metrics:

We use metrics listed below to evaluate the different aspects of Sesame:

**Average Response Time:** We present the average and standard deviation of the query response time across the three runs. (*Error bars are shown, but may not be visible as the coefficient of variation is under 3% for all measurements.*)

**Cache Hit Rate:** The Cache Hit Rate is defined as the fraction of the user queries answered from the query cache. It gives a sense of the effectiveness in predicting the next user query. It is also dependent on the speculation duration.

**Speculation Hit Rate:** Speculation Hit Rate is defined as the ratio of speculative queries that are reused by the user query to the total number of speculative queries run.

**Overall Speculation Benefit:** Running speculative queries results in query speedup at the additional cost of having to execute them. Hence, we devise a metric that takes both these factors into account to give us an overall idea about the benefits of speculation. We define the metric Overall Speculation Benefit as Query Speedup \( \times \) Speculation Hit Rate, where Query Speedup is defined as the ratio of the execution time under the naive case (no speculation) to that under Sesame. The benefits of speculation might be considered to outweigh its cost when this metric is greater than 1.

3.5.3 Results

We define \( \text{ALGO}_{\text{FULL, SPEC}} \) as the algorithm without any limits on speculation duration. As a result, all user queries can be answered from the cache. The algorithm in which the speculative queries are chosen, as given in Section 3.4.1, and run within a bounded time interval is termed as \( \text{ALGO}_{\text{SESAME}} \). Finally, we term \( \text{ALGO}_{\text{NAIVE}} \) as the algorithm where the speculative query execution is turned off. Thus, comparing \( \text{ALGO}_{\text{FULL, SPEC}} \) to
ALGO\textsubscript{NAIVE} gives us an idea about the benefits of speculation. Comparing ALGO\textsubscript{FULL\_SPEC} to ALGO\textsubscript{SESAME} informs us about the detrimental effects of not answering a query from the query cache. During the experiments, the default maximum speculation duration was set to 30 seconds. Note that in the graphs, the number of shards increases at an exponential rate. Due to the fact that the data is horizontally partitioned into shards, once the queries are run on individual shards, results can be combined to present a single result. This is an expensive step and common to all three algorithms described above. However, since data aggregation is not the focus of this work, and we want to better illustrate the benefits of speculative execution, the results containing data aggregation are presented only in Sections 3.5.3.7 and 3.5.3.8.

3.5.3.1 Response Time across Varying Data Sizes

![Figure 3.4: Different Dataset Sizes (Workload\textsubscript{TPCDS})](image)

Execution times for both ALGO\textsubscript{NAIVE} and ALGO\textsubscript{FULL\_SPEC} increase linearly with the increasing size of datasets for both workloads (Figures 3.4, 3.5). The benefit of our system
Figure 3.5: Different Dataset Sizes ($Workload_{Real}$)

is evident: $ALGO_{SESAME}$ is typically at least an order of magnitude faster than traditional database querying. As an example, with 192 shards, $ALGO_{SESAME}$ is $18 \times$ faster than traditional execution for $Workload_{TPCDS}$ and $25 \times$ faster for $Workload_{Real}$.

As expected, the execution time for $ALGO_{SESAME}$ increases linearly until some of the queries need to be run on the underlying data instead of on the cached materialized views. To illustrate the importance of retrieving results from the cached materialized views, even when 99% of the queries can be answered from the cached views, speedup falls from 25.77 (for 100% Cache Hit Rate) to 21.35, and further to 9.45 when the Cache Hit Rate was 91% (Figure 3.5).

### 3.5.3.2 Overall Speculation Benefit

The Overall Speculation Benefit is measured across varying dataset sizes (Figure 3.6). This metric follows the query speedup, which increases initially as a greater fraction of time is spent actually running the query compared to the time spent in setting up the query execution, whose increase is comparatively slower. Notably, the Overall Speculation Benefit is consistently greater than 1. As the sampling rate increases, a smaller fraction of the
queries can be answered from the cache, due to which the metric starts decreasing. Thus, we can use this metric to evaluate whether speculation is worthwhile.

### 3.5.3.3 Response Time across Varying Speculation Durations

![Graph showing the effect of speculation duration on response time](image)

**Figure 3.7: Effect of Speculation Duration on Response Time**
As the speculation duration decreases beyond a threshold, the execution time starts increasing (Figure 3.7). Until this threshold, all results could be answered from the cache. Beyond this point, a greater fraction of the user query will need to be run on the underlying data as opposed to the materialized views, which can be orders of magnitude more expensive.

### 3.5.3.4 Cache Hit Rate Across Differing Speculation Durations

![Graph showing the effect of speculation duration on cache hit rate](image)

Figure 3.8: Effect of Speculation Duration on *Cache Hit Rate*

As the speculation duration decreases beyond the same threshold as above, the *Cache Hit Rate* also linearly decreases (Figure 3.8). The linear decrease is expected since the number of queries that can be speculatively run will decrease linearly as well, thereby resulting in a corresponding drop in the *Cache Hit Rate*. 
3.5.3.5 Comparison of View Sizes Needed

Since $Sesame$ requires materializing additional speculative views, we investigate the space costs associated with both $ALGO_{NAIVE}$ and $ALGO_{SESAME}$ (Figure 3.9). Note that both of these techniques need to materialize views at the end of the user query running on individual shards in order to aggregate the results into a single result set. We can see that $ALGO_{SESAME}$ requires a maximum of six times the space as $ALGO_{NAIVE}$. Considering the benefits of improved response time and the low/plummeting costs of storage, this added expense is clearly worth the query speedups in the context of interactive execution.

![Figure 3.9: Comparison of View Sizes](image)

3.5.3.6 Comparison of $Variance_{Fast}$ and $Variance_{Common}$

In Section 3.3.2, we have discussed the reasons behind $Variance_{Fast}$ giving results much sooner than $Variance_{Common}$. With increasing data size, a greater fraction of the queries need to be computed by accessing the underlying data and the benefits of using a faster
variance evaluation technique become even more prominent. This is shown by the increasing speedups with increasing data size (Figure 3.10) offered by $\text{Variance}_{\text{Fast}}$ as compared with $\text{Variance}_{\text{Common}}$.

This experiment was designed to study the effect of processing increasing amount of data on the execution time. While the benefit of $\text{Variance}_{\text{Fast}}$ was more apparent at higher sampling rates (384 and 768 shards), another way to look at the experiment would be to consider the different number of shards processed to represent a small (say 1%) sample of a larger dataset. In that case, the benefits of using the faster representation would be apparent at low sampling rates as well.

It is well-known that even though $\text{Variance}_{\text{Fast}}$ is used in multiple systems such as PostgreSQL, System X, etc. [78], it can suffer from numerical precision issues. Hence, we tested the quality of the results when using $\text{Variance}_{\text{Fast}}$. We used $\text{Variance}_{\text{Common}}$ as the ground truth. Our results showed that the median relative difference in the results was low, around $O(10^{-4})$, for all sampling rates for (Workload$_{\text{Real}}$).
3.5.3.7 Hotspots – Where is Time Spent in Sampled Aggregations?

We investigate the distribution of query execution time amongst the different phases of sampled aggregations (Figure 3.11). Before actually running the queries, we formulate a plan for doing so, taking into consideration the already cached results. We term the time spent doing this as $Time_{PreQuery}$. Next, we define the time spent actually running the queries on the shards as $Time_{Query}$. Finally, we denote by $Time_{PostQuery}$, the time spent aggregating the results from the individual shards into a single combined result. We can see that $Time_{PostQuery}$ is an expensive step. We can also note that $Time_{PreQuery}$ is negligible compared with others.

![Figure 3.11: Distribution of Execution Time in Sampled Aggregations](image)

3.5.3.8 Naive Aggregation vs Aggregation Tree

As mentioned above, once the queries are run on individual shards, their results need to be combined to present a single result to the user. In doing so, an obvious solution is to
union all results and then combine them \( (ALGO_{\text{NAIVE\_AGGREGATION}}) \). Another option is to combine the results using an aggregation tree \( (ALGO_{\text{TREE\_AGGREGATION}}) \). In Figure 3.12, we can see that \( ALGO_{\text{TREE\_AGGREGATION}} \) plays well with \( \text{Sesame} \) and gives results quicker than \( ALGO_{\text{NAIVE\_AGGREGATION}} \).

![Graph](image_url)

Figure 3.12: Naive Aggregation vs Aggregation Tree

### 3.5.3.9 Impact of Cores

We performed a scale-up experiment where we increased the number of cores, keeping other factors the same (Figure 3.13). We can see that performance improves upon increasing the number of cores. The time taken for non-parallelizable code is trivial compared with the time taken to actually run the queries. The reason for the non-linear increase in speed-up with increasing number of cores is that the scale-up is only with regards to the number of cores – other hardware aspects such as memory, bus stay the same. Hence, with a linear increase in the number of cores, the speedup cannot be expected to be linear.
This experiment inspires us towards an interesting observation. The execution time while using only 6 cores using Sesame’s speculation is far lesser than the execution time using all the 48 cores without any speculation. For achieving speedups, a user can use more hardware and shard the data across multiple machines, which results in additional costs. However, we can see that without resorting to improving the hardware, we are able to improve query latencies. Thus, speculation can be considered as a cost-saving measure in some circumstances (load on the server is low).

![Figure 3.13: Effect of Increasing Number of Cores (Workload\textsubscript{Real})](image)

3.5.3.10 Impact of Multi-user workloads

We have so far demonstrated the value of Sesame from a single user’s perspective. However, the speculation techniques mentioned in this work are applicable to multi-user contexts as well. We used 5 query sessions based on the exploratory queries of the graduate students as described earlier, each consisting of 10 queries. Processing queries from multiple users simultaneously presents opportunities for cross-user cache reuse for both the user queries as well as speculative queries.
We can see that speculative queries take around 20% lesser time. The improvement in latencies of user queries varies from 3% to 19% for 384 shards. The reason for the improved performance of user queries for 384 shards is that not all results at this sampling rate can be retrieved from the cache in the case of the queries being run separately. However, when the queries and speculative queries for multiple users are run as part of the same session, it is possible to leverage the common speculative queries across multiple users and run more speculative queries as a result. This results in a larger fraction of the user queries being answered from the cache giving better latencies.

Figure 3.14: Effect of Multiple Users (Workload\textsubscript{Real})

3.5.3.11 Distributed Execution Backend

One benefit of having a rewrite-based infrastructure is the ability to study the performance of \textit{Sesame} in a distributed environment. Figure 3.15 shows the performance comparison of \textit{Sesame} using Cloudera Impala 1.2.4 (latest available version on Amazon EC2 EMR) as the backend using 1 coordinator and 15 worker nodes on Amazon EC2 of type \texttt{i2.2xlarge}
each having 8 cores (2.5 GHz), 61GB of memory and $2 \times 800$ GB of SSD with the maximum speculation duration set at 90 seconds. We can see that Sesame scales well in a distributed environment giving speedups of up to $4.03 \times$. While these benefits are clearly significant, they are not as high as that of single-node PostgreSQL – having a distributed backend impacts Sesame’s caching layer as well, which introduces additional latency and overhead. Architectural optimizations for distributed contexts would be ideal future work.

![Graph showing execution time vs number of shards.](image)

**Figure 3.15: Sesame on Impala (WorkloadReal)**

### 3.5.3.12 Comparison with BlinkDB

It should be noted that Sesame’s contributions are towards error reuse and speculation strategies focusing on session-based sampled aggregation workloads, and are strictly orthogonal to various offline sampling techniques such as those in BlinkDB. These systems can always be combined (with considerable engineering effort) to yield additional benefits: incremental samples used in BlinkDB can be mapped to our shards, and the online sample selection strategy can be carefully integrated with ours.
Figure 3.16 shows a performance comparison of Sesame and BlinkDB (available at http://blinkdb.org). Both Sesame and BlinkDB were run on Spark 1.1.1 over the same sized data in the distributed environment described earlier (16 i2.2xlarge nodes on Amazon EC2, with a maximum speculation duration of 90 seconds). We can see that both systems perform comparably with increasing data size. It should be noted that while BlinkDB estimates variance by using the error profile over a smaller subset, Sesame computes the exact variance over the entire queried sample. Additionally, Sesame’s performance can be improved further by tuning it specifically for Spark, as opposed to our current backend-agnostic approach.

![Figure 3.16: Comparison against BlinkDB (WorkloadReal)](image)

### 3.5.4 Insights

We have seen the benefits of reusing the variance computation to speed up query execution using speculative querying and have studied the benefits offered by each of Sesame’s features. A noticeable insight is the high penalty paid initially when the Cache
Hit Rate falls below 100%, which in fact demonstrates the benefits of reusing variance computation. Upon successive decrements of the Cache Hit Rate, the query response time increases linearly. This is explained by the fact that the overall response time is governed by the time to execute all the queries. Having an initial drop in Cache Hit Rate results in higher response times of a few queries, bumping up the overall response time. We believe we can mitigate these effects by improving our query scheduling mechanism by scheduling queries that need to access underlying data before those that can reuse the results.

The variation in query response time of Sesame is quite low. This can be evidenced by the fact that the coefficient of variation is at most 3% for ALGONAIIVE, ALGOSESAME, and ALGOFULL_SPEC for both datasets. For this reason, the error bars are not clearly visible, though they are displayed in most cases.

Studies by Liu et al. [95] have shown that adding a 500 ms additional delay results in significant costs with regards to decreased user activity and dataset coverage. Sesame provides speedups of up to 25X – far greater than the 33% offered by DICE, in which the user studies have demonstrated noticeable benefits of speedups offered for interactive analytics settings. Further, the speedups offered by Sesame result in a significant and visually noticeable reduction in response time from an order of multiple seconds to milliseconds.

3.6 Related Work

Data Cubing: Data cubes after being proposed in the seminal work by Gray et al. have since found wide use in diverse contexts, including peer-to-peer systems [76]. Choosing views and indexes given constraints such as space and maintenance cost has been studied before [6, 65, 125]. Materializing samples [92] and succinct representations [134] of the cube have also been investigated.
View Reuse: There has been significant work in query rewriting using materialized views \([34, 61, 100, 116]\) for general purpose, warehousing, and interactive contexts; *Sesame* draws from this body of work to design our session and sampling-aware caching strategies. In *Sesame*, we articulate the various choices for view selection in an online setup in a session-aware manner and provide suggestions to choose the fastest option. DynaMat \([87]\) presents view selection and maintenance under a unified framework and materializes an optimal set of views. While the goal of *Sesame* is similar, our focus is on sampled aggregations in the context of a session, taking into account user retention as well.

Approximate Querying and Data Sampling: Generating samples under constraints such as space \([2]\), bounded-size sampling schemes \([53]\), etc. has also been researched. \([111]\) studies sampling in the context of joins. Sampling has also been intrinsically incorporated in database systems \([86]\) and distributed systems \([28]\). Since *Sesame* works by performing query rewrites, any SQL-based backend can be plugged into our system, allowing us to leverage their features.

Prefetching: Prefetching has long been used to provide query speedups at various levels of the computational stack such as caching \([10, 32]\), analytical queries \([41]\), spatial data \([142]\), etc. Caching and prefetching analytics query results based on the user’s query patterns has been considered before \([15, 77, 160]\). However, in the context of sampled aggregations, the computation of error has always been a bottleneck, which our work addresses.

3.7 Conclusion & Future Work

We have presented *Sesame*, a novel multi-threaded framework that leverages holistic optimization of the user query session to reduce query response time by up to an order of magnitude when performing aggregations over sampled data. Reusing variance computation
helps avoid processing the entire data to calculate variance. We can use result sets of already computed queries, which can be smaller by multiple orders of magnitude, to do so. The reduced data size of these results increases the likelihood of them fitting into various levels of cache and memory hierarchy, further speeding up query execution. *Sesame* is the first work to look into the algebraic properties of *variance* as part of its cache reuse mechanism. We show how using different representations of variance affects query response times and suggest using the representation that provides the fastest result in the context of interactive query execution. We note that speculative query execution depends primarily on two factors – speculation accuracy and speculation duration. Speculation duration is dependent on the time taken by the user to peruse the results and is beyond the system developer’s purview. Speculation efficiency depends on the user behavior and the query model and can be improved with better query logs that reflect the current user’s behavior.

A complementary next step would be the investigation of faster techniques for post-aggregation from the different shards since this is an important and expensive step in sampled aggregation. We plan to look into database backends that support hierarchical aggregation trees, multi-query optimization, and query pipelining approaches for this.

Another major avenue of future work for *Sesame* will be user guidance. Speculative query execution results in parts of the cube near the current user query being computed. This can be considered as a partial cube materialization strategy. Useful information can be provided to the user by mining the partially materialized cube to discover facts such as interesting groups [129]. Further, finding that a region has a higher interestingness quotient can let us devise strategies such as speculatively executing queries belonging to it at a higher sampling rate, which can also help guidance and query steering systems such as *AIDE* [41].

76
Thus, *Sesame* improves on the result reuse framework of *DICE* by extending reuse to errors. It uses query rewriting and materialized view generation to further speed-up query results. It provides novel strategies for both view selection as well as shard selection.
Chapter 4: Combining User Interaction, Speculative Query Execution and Sampling in a Frontend

4.1 Introduction

Current database frontends are not designed taking into consideration the session-oriented behavior of the user in OLAP scenarios. A backend such as DICE, which facilitates query execution using interdependence of cubing queries is greatly benefited by a frontend that does so as well. The DICE frontend is therefore co-designed with the faceted exploration-based session-oriented backend [70]. This not only helps guide the user but also improves the speculation efficiency. The user is given an option to navigate using either the user-friendly point-and-click approach or the more powerful way of composing the query themselves. Techniques such as these make our frontend ideal for session-oriented backends such as DICE.

4.2 System Overview

We present an overview of the user interaction aspects of the DICE system, detailed in Section 2. The hierarchical coordinator-worker architecture based backend is connected to the frontend UI. All queries are issued to the coordinator by the UI and responded to by the coordinator. The coordinator, in turn, distributes these queries amongst the worker
nodes. Given a user query, DICE enumerates possible speculative queries using the *faceted exploration model* (described in Section 4.3) and calculates their corresponding likelihoods from the query logs. Due to the scale of the data and the number of speculative queries, it is not possible to execute all speculative queries at the highest sampling rate. Hence, we choose a subset of queries to execute and cache their results to maximize the probability of the next user query being present in the query cache.

**Figure 4.1: DICE User Interface Query Flow**

**Query Flow:** Figure 4.1 describes the overall query flow of DICE in terms of the sequence in which the user query, accuracy queries, and the speculative queries are run. The user query is run as soon as the user requests for it. After receiving the query result, the *accuracy* queries are run with the additional measures of *COUNT*, *SUM*, and *VARIANCE* which are
needed to estimate the combined sampling accuracy for the measures MEAN and SUM. Upon execution of the accuracy queries, speculative queries are scheduled.

4.3 Faceted Exploration Model and Frontend

For any aggregation in the data cube, the number of possible follow-up queries is extremely large. This creates a challenge for our speculative execution technique – which queries do we pick? Thus, we present a model where the user explores the cube in a series of *faceted traversals*, thereby, restricting the space of queries. The *faceted cube exploration* model is used to guide user exploration at the frontend, and also to speculate and execute a subset of the next possible queries at different sampling rates so as to maximize the probability of the next query being present in the query cache at a high sampling rate. This synergistic model enables us to design an intuitive and easy to use interface to explore the data cube and, at the same time, reduce response time using speculative execution.

4.4 User Experience

4.4.1 System Design Principles

We built the frontend and the backend of the *DICE* system based on the following design principles. Unlike typical systems, the query model assumes query sessions, interactivity, sampling, and speculation as key concepts.

**Faceted Exploration:** User interaction is strongly aligned with the faceted model. Exposing faceted traversals as UI elements has been shown to be effective in data cube traversal [77]. Faceted exploration helps inform the user about their location in the data cube and helps enumerate the neighboring facets, which make up the set of speculative queries. While this aids the user in cube traversal, it also allows us to execute a subset of
Figure 4.2: DICE User Interface that allows for interactive, intuitive cube exploration.

the speculative queries during the result perusal time, thus reducing the latency of the user query, and enhancing the synergy between user guidance and query latency reduction.

**Latency vs Accuracy:** Users should be able to interactively adjust sampling rates based on their needs, possibly resulting in lower latency but higher errors. This is key for interactive cube exploration since depending on the number of sampled data points, data skew, the selectivity of the WHERE predicate, and the user desired accuracy, the sampling rate should be allowed to vary, giving greater control over analysis to the user.

**Results First:** The results of a query should be prioritized and shown first, over the estimation of errors. Running the accuracy queries which consists of additional measures added to the user query results into an additional time expense of around 20%. As is evident from the query flow, we give greater importance to faster results compared to the accuracy estimation and hence, run the accuracy queries after the initial user query is executed. On
the user interface, this is presented as large circles in the scatter plot, which later resolve into smaller points with error bars.

4.4.2 User Interface

We now describe various parts of the DICE frontend user interface (Figure 4.2). As described in the design principles, the query interface is inspired by the redefined sampling-aware query paradigm and the speculative execution capabilities of the backend.

**Configuration:** In the top left corner of the UI, a user can specify the sampling rate, the choice of whether to use speculative execution (Algorithm: DICE) or not (Algorithm: NOSPEC).

**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 faceted traversals in the lattice, surfacing followup queries that are likely to be speculated.

**Interactive Query Editor (IQE):** One way to use the query editor is simply to enter the SQL query in the text box. Another more intuitive, easy-to-use option is provided below it in the form of the *Traversal* button. Clicking on it provides options to add a dimension resulting in a *child* traversal; remove a *WHERE* predicate causing a *parent* traversal; change the value of a *WHERE* predicate, thereby, inducing a *sibling* traversal; or swap a dimension in the case of a *pivot* traversal. In the case that the *WHERE* predicate is to be changed, the system will prompt the user with the range of values for the dimension. This allows the user to edit the query without advanced knowledge of the query language or schema. After the changes have been made to the SQL query, the query is automatically executed. Thus, the *IQE* allows the user to perform a *full* faceted cube exploration by providing different traversal options
in a visual manner without having to type any part of the SQL query, enabling the user to navigate the entire data cube in a series of clicks. It is up to the user to use the more visual nature of the IQE or participate in a more hands-on fashion by entering the query themselves. Thus, the IQE draws on the faceted model to enhance the session-oriented behavior of the user.

**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 interactive features of the *Editor* are used, there is no need for the *Execute* button to be clicked.

**Timing Statistics:** As a diagnostic aid, DICE also shows the 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 to the user about split up of the execution costs as well as how well the system is able to model user behavior.

**Histogram Viewer:** Results are presented at the bottom. The viewer displays a scatter plot as soon as possible to the user, maximizing utility. As discussed in Section 4.2, since queries are typically performed over sampled data, the system then issues a secondary set of queries (accuracy queries) to infer the error estimates, which it overlays as the 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.

### 4.5 Related Work

Numerous business intelligence tools [14, 22, 63, 157] provide visualization and interactive interfaces to large multidimensional datasets. Cetintemel et al. [41] provide a vision for a system to guide the user in interactive querying. Techniques in online aggregation [69] have similar motivations to ours but do not consider the influence of the interaction on
system design. *DICE*, on the other hand, uses an online sampling approach to get the query results and accuracy specifically targeted towards faceted cube exploration and as such is complementary to it. BlinkDB [4] considers only single queries (as opposed to query sessions) and computes an offline sample of numerous column combinations at multiple resolutions. Tools such as Tableau [63] translate visual interactions into a series of SQL queries, and interactive loops correlate directly with our session-based model. As discussed before, we observe that such interactions directly correspond to facet traversals, allowing us to utilize actual workloads from such tools in our experimental evaluation.

### 4.6 Conclusion & Future Work

We introduced the idea of simultaneously designing interaction and database architecture in *DICE*, a distributed, interactive cube exploration system. *DICE* provides a natural and intuitive way of faceted exploration of data cubes. The visually enhanced and easy point-and-click interface provided by *DICE*’s frontend helps not only novice users but also expert users by guiding them through the complexities of the data cube in a faceted exploration model. While typical data warehouses are designed around improving latencies for the user query, using the fact that in OLAP scenarios, queries usually don’t exist in isolation but as part of sessions, we can provide improved latencies by using speculative execution. The frontend helps speculation by considering faceted traversals while suggesting the next query to the user. We can see that our frontend is able to use the features and strengths of our backend – indeed to fully utilize the backend properties, we have shown that the frontend should be built synergistically. Our frontend can also be plugged into *Sesame* without a lot of effort, demonstrating the orthogonality between the frontend and backends as well as the conceptual similarity between the two backends.
Chapter 5: A Unified Correlation-based Approach to Join Sampling

5.1 Introduction

In cases where the data size restricts the hardware or software’s ability to process it within a reasonable time, sampling presents a pragmatic approach towards providing insights at scale. Joining multiple datasets helps incorporate related knowledge sources and develop deeper insights. Performing statistical analyses using aggregations is a popular step following the joins. Sampling over joins is a compelling, yet challenging task. Performing a join can be expensive – sampling after materializing the join may not be a pragmatic approach. Initial efforts in join sampling were directed towards obtaining non-correlated samples\(^2\) [3, 31, 113]. As Chaudhuri et al. [31] demonstrated the inherent hardness of the problem and as online aggregation [69] started gaining momentum, efforts came to be directed towards obtaining correlated, probability samples for aggregation queries [44, 59, 71, 83, 149]. However, non-aggregation use cases (such as presenting a sample to the user) have not been considered by this line of work. Correlated samples usually also have higher error estimates compared with non-correlated samples [37]. In this context, we aim to reduce correlation in correlated samples by maximizing join randomness (number of possible

\(^2\)We define a correlated sample as one where the inclusion probability of an item depends on that of another (details in Section 5.1.1.1) – samples having any degree of randomness in the sampling process can be called random samples [37].
samples as a result of a join algorithm) – this has a side-effect of lower sampling error when compared with other correlated sampling techniques. Further, in the case that non-correlated samples are mandated, we suggest enhancements to the state-of-the-art algorithms. To better understand the problem domain, we illustrate the primary challenge in the non-correlated sampling of joins using an example.

5.1.1 Challenges in Avoiding Sample Correlation

Chaudhuri et al. [31] and Gibbons et al. [3] have provided excellent examples demonstrating one of the difficulties of join sampling – a relation has to be sampled in a biased fashion while considering the join key cardinality of the other. We look at another key challenge – that of the sample size far exceeding the relation size. Before doing so, we first demonstrate the cause of sample correlation, avoidance of which results in the aforementioned challenge.

5.1.1.1 Correlated Samples

To avoid correlation, a tuple being present in a sample should not affect the inclusion probability of another. For example, consider samples $S_1 = \{t_1, t_2\}$ and $S_2 = \{t_3, t_4\}$ of relations $R_1$ and $R_2$ respectively. Join between $S_1$ and $S_2$ produces tuples $\{t_1 \circ t_3, t_1 \circ t_4, t_2 \circ t_3, t_2 \circ t_4\}$, where $t_i \circ t_j$ represents concatenation of $t_i$ and $t_j$. Each tuple in $S_1$ and $S_2$ results in multiple join tuples, i.e., presence of a tuple, $t_i \circ t_j$, in the output increases the probability of other tuples having their $R_1$ component be $t_i$ or $R_2$ component be $t_j$, resulting in a correlated sample.

5.1.1.2 Sample Inflation

We illustrate how avoidance of correlation can result in large samples. Consider joining two relations, each having a single key, and cardinalities of 50 and 100 respectively, causing
join to have 5000 tuples. A 0.1 fraction of the join will consist of 500 tuples – far larger than the size of either relation.

If a sampled tuple produces multiple joined tuples, the tuples will be correlated. To avoid correlation, a sampled tuple has to be restricted to join only once. This results in both samples having a size of 500, making sampling counter-productive. We define this need to avoid correlation between output tuples resulting in increased sample size as sample inflation. It is the primary reason behind sampling over joins being difficult, and an infeasible proposition at times. While previous works provided algorithms for sampling joins [3, 31], ours is the first work to formalize the inherent difficulty of join sampling.

5.1.1.3 Sample Inflation in Current Non-Correlated Sampling Approaches

Chaudhuri et al. [31] provide algorithms to obtain non-correlated samples for different index and statistics availabilities. We demonstrate how sample inflation can occur in each of these algorithms – looking at their experimental results, it appears that they avoided sample inflation through either careful implementation or query plan optimization. We provide enhancements to these non-correlated sampling algorithms to reduce the sample size and intermediate data size. However, the reduction might not be satisfactory due to sample inflation. Hence, the primary focus of our work is to provide algorithms to reduce the correlation in correlated samples by maximizing the number of possible samples. This is based on our observation that non-correlated sampling techniques result in the maximum possible number of samples. Our efforts, thereby, in increasing the number of possible samples are directed towards making the samples as non-correlated as possible.
5.1.2 Overview & Contributions

We provide two key contributions – first, we look at techniques to maximize the number of possible samples in correlated sampling under the constraint of fixed sample size, when statistics over the join column are available. We provide strategies for allocating samples to different strata of multiple relations for different join types, including equi-join, outer join, self-join, non-equi-join, and theta join for the comparators $<$, $\leq$, $>$, $\geq$ (Sections 5.2 and 5.3). These techniques are derived mathematically in Section 5.8. Although the derivations are complex, the resultant allocation strategies are simple, intuitive, and easy to compute. They have been experimentally validated to provide allocation close to the optimal allocation that was found using a brute-force search. The sampling error of our techniques was found to be lower than that of other correlated join sampling techniques (Section 5.5.2.2).

Our second contribution is in non-correlated sampling, where we provide enhancements to the state-of-the-art algorithms [31]. When complete or partial statistics are available over a relation, our algorithms, GROUP-SAMPLE-ENHANCED (Section 5.4.1), and FREQUENCY-PARTITION-SAMPLE-ENHANCED (Section 5.4.2), access only half of the tuples in comparison, avoid the need to create large intermediate data which can be larger than the base relations, and remove a sampling step over the intermediate data. In the case where statistics and indexes are available over a single relation, we specify a filter-based criterion to decide whether sampling should be used, violating which can result in the sample size exceeding the relation size (Section 5.4.3). We also use this criterion to sample both relations, when statistics and indexes are available over both (Section 5.4.4).
5.2 Maximizing Randomness For Equi-Join

As mentioned earlier, we approach the correlated sampling problem from the perspective of maximizing join randomness. The problem statement we address can be given as follows:

*For the case where correlated samples are acceptable, determine the sample allocation strategy to maximize join randomness, given the constraint of fixed sample size, in the presence of statistics over join columns.*

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_i$</td>
<td>$i^{th}$ Relation of the Join</td>
</tr>
<tr>
<td>$N_i$</td>
<td>Cardinality of $R_i$</td>
</tr>
<tr>
<td>$S_i$</td>
<td>Sample of $R_i$</td>
</tr>
<tr>
<td>$n_i$</td>
<td>Cardinality of $S_i$</td>
</tr>
<tr>
<td>$f$</td>
<td>Join Sampling Rate</td>
</tr>
<tr>
<td>$A$</td>
<td>Join Column</td>
</tr>
<tr>
<td>$a$</td>
<td>Join Value</td>
</tr>
<tr>
<td>$t.A$</td>
<td>Value of Column $A$ in Tuple $t$</td>
</tr>
<tr>
<td>$m_i(a)$</td>
<td>Number of Tuples in $R_i$ having Value $a$ in $A$</td>
</tr>
<tr>
<td>$m^j_i$</td>
<td>Number of Tuples in $R_i$ belonging to $j^{th}$ stratum</td>
</tr>
<tr>
<td>$mm_i(a)$</td>
<td>Number of Tuples in $S_i$ having Value $a$ in $A$</td>
</tr>
<tr>
<td>$mm^j_i$</td>
<td>Number of Tuples in $S_i$ belonging to $j^{th}$ stratum</td>
</tr>
<tr>
<td>$z$</td>
<td>Number of Relations</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of Strata</td>
</tr>
<tr>
<td>$k$</td>
<td>Total Sample Size</td>
</tr>
<tr>
<td>$k^j$</td>
<td>Number of Samples Allocated for Stratum $j$ for all Relations</td>
</tr>
</tbody>
</table>

Table 5.1: List of Notations

5.2.1 Join Randomness

We use *join randomness*, defined as number of samples possible in a join algorithm, to reduce correlation in a sample. To understand our motivation behind doing so, let us look at an example depicting the number of possible samples as a result of a few different sample
allocations. Consider joining two relations with strata sizes, \( m_1^1 = 10, m_1^2 = 10, m_2^1 = 20, \) and \( m_2^2 = 20. \) Let the sample size be constrained at 30. The number of possible samples is 
\[
C_{mm_1^1}^{m_1^1} \times C_{mm_2^1}^{m_1^2} \times C_{mm_1^2}^{m_2^1} \times C_{mm_2^2}^{m_2^2}.
\]
Table 5.2 shows that different sample allocations can result in the number of possible samples differing by multiple orders of magnitude.

<table>
<thead>
<tr>
<th>Index</th>
<th>( m_1^1 )</th>
<th>( m_1^2 )</th>
<th>( m_2^1 )</th>
<th>( m_2^2 )</th>
<th># Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>( 2.2 \times 10^{15} )</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>12</td>
<td>( 2.3 \times 10^{14} )</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>12</td>
<td>13</td>
<td>( 5.3 \times 10^{13} )</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td>14</td>
<td>( 1.5 \times 10^{11} )</td>
</tr>
</tbody>
</table>

Table 5.2: Effect of Sample Allocation on Number of Samples

If we are not restricted to sampling the input and then performing the join but could sample the join of the relations (resulting in a non-correlated sample), the number of possible samples would have been exceedingly large, 
\[
C_{|R_1 \bowtie R_2|}^{500} = C_{|S_1 \bowtie S_2|}^{125}
\]
for the first configuration. Thus, our efforts can be perceived as being towards making the samples as non-correlated as possible, through the metric of the number of samples. This view has been strengthened by our experiments, which show that our approach has a lower error than other correlated sampling-based approaches – non-correlated samples result in lower error than correlated samples theoretically.

### 5.2.2 Maximizing Randomness for Single Stratum

Consider allocating \( k^j \) tuples amongst relations \( R_1, R_2, \ldots, R_z \), each having a single stratum, to maximize the number of possible samples, 
\[
\prod_{i=1}^z C_{mm_i^j}^{m_i^j}.
\]
Section 5.8.1 shows that Equation 5.1 can be used for this purpose – Section 5.5.1.1 shows that it results in a low error, with the maximum difference in the sample stratum values as a result of our allocation.

90
and the optimal allocation found by searching through all possible allocations being 2.

\[
mm^j_i = \text{round} \left( \frac{k^j \times m^j_i}{\sum_{i=1}^{n_j} m^j_i} \right)
\]  \hspace{1cm} (5.1)

### 5.2.3 Maximizing Randomness for Multiple Strata

We now provide the strategy to allocate a given sample size amongst different strata, in the general case of equi-join between multiple relations having multiple strata. Consider the problem of determining the sample allocation \(k^j\) for \(j^{th}\) stratum, for \(j \in [1, n]\), with \(k = \sum_{j=1}^{n} k^j\). Our goal is to maximize the number of possible samples, \(\prod_{j=1}^{n} \prod_{i=1}^{m} C_{mm^j_i}^{m^j_i}\). Section 5.8.2 shows that we can use Equation 5.2 to do so – Section 5.5.1.2 shows that the allocation is close to optimal allocation, with the maximum difference being 1.

\[
k^j = \text{round} \left( \frac{k \times \sum_{i=1}^{z} m^j_i}{\sum_{j=1}^{n} \sum_{i=1}^{z} m^j_i} \right)
\]  \hspace{1cm} (5.2)

### 5.2.4 Combined Allocation – MAXRANDJOIN

Using equations 5.1 and 5.2, \(mm^j_i\) can be given as follows.

\[
mm^j_i = \frac{m^j_i}{\sum_{i=1}^{z} m^j_i} \times \frac{k \times \sum_{i=1}^{z} m^j_i}{\sum_{j=1}^{n} \sum_{i=1}^{z} m^j_i} = \frac{k \times m^j_i}{\sum_{j=1}^{n} \sum_{i=1}^{z} m^j_i}
\]  \hspace{1cm} (5.3)

### 5.2.5 Derivation of Allocation Strategy

We use Lagrange multipliers, a popular tool in the statistical sampling community to find approximate optimal strata allocation in closed-form under space or cost constraints [96]. This includes different approaches for determining strata sizes in stratified random sampling such as Neyman allocation and cost-based allocation [96]. Lagrange multipliers provide us with critical points for maximum and minimum values if they exist. Our functions will have a minimum value as the number of samples is non-negative. They will also possess a
maximum value as the relation and sample sizes are bounded. The critical points have to be plugged into the function for the number of possible samples to determine if the resultant value is a maximum or a minimum. Our experiments (Section 5.5.1) show that suggested sample sizes are close to the optimal solution in practice – with a maximum difference of 2 in the case of single stratum allocation and 1 for multiple strata allocation. The derivations possess a couple of sources of potential error. They might result in rounding errors as they maximize the number of samples in the continuous domain, whereas the allocation occurs in the discrete domain. Another source of error can be our use of an approximation for the harmonic sum, which results in the assumption of \( m m_i \) and \( (m_i - mm_i) \) being large (Equation 5.10). Finally, we note that our simple, intuitive, closed-form formulae maximize expressions consisting of factorials that are combinatorial in nature.

5.2.6 Comparison with Correlated Join Samplers

Our approach embraces correlation in the samples and aims to reduce it by maximizing the join randomness. Section 5.5.2.2 shows that a side-effect of maximizing the join randomness is lower sampling error when compared with other correlated samplers.

End-Biased Sampling, Correlated Sampling, and Universe Sampler provide a cluster sample – they are ill-suited for handling correlation between join and measure columns. Such approaches circumvent the sample inflation issue by considering only those tuples that their hashing function accepts. This removes the need for histogram information as well. On the other hand, we provide a unified approach to both correlated and non-correlated sampling scenarios using strategies with sound mathematical origins that tackle sampling inflation head-on.

92
Ripple Join, SMS Join, and Wander Join, at every point in their execution, provide a correlated output as a result of a join between simple random samples of the relations. These approaches do not take strata-based skew into consideration and as a result have a higher error than our approach (Section 5.5.2.2).

MAXRANDJOIN is specifically designed to provide a sample that is as random as possible and takes all strata into consideration. In contrast, other correlated samplers have different goals such as responsiveness, streamability, scalability, removing the need for statistics and indexes, etc. Comparing MAXRANDJOIN to them is not straightforward – they have different objectives with the resultant benefits and drawbacks.

5.2.7 Other Allocation Strategies

Our proportional allocation-based strategy depends only on the strata sizes of join columns. It does not need any other information. It might be possible to extend the algorithm to make it optimal from variance and sampling cost perspectives as well.

Using variance has the drawback of knowing information about the measure columns beforehand or performing another pass over the data. Further, our strategy is independent of the applications that the sample might be used for while using the variance of the measure column restricts us to the aggregation use case. The case of distinct sampling costs introduces a dependency on system information. These might be problems to look at in the future – they are beyond the scope of the current work that only depends on join column statistics and produces a sample from an application-agnostic perspective.
5.3 Maximizing Randomness for Other Join Types

We have looked at maximizing randomness for the important case of equi-joins. We build upon the results and mathematical tools developed in the previous section to present techniques for maximizing randomness for all other join types.

5.3.1 Outer Join & Self-Join

The expressions for the number of samples that need to be maximized in the case of equi-join, \( \prod_{i=1}^{z} C_{mm_i}^m \) (for single stratum) and \( \prod_{j=1}^{n} \prod_{i=1}^{z} C_{mm_i}^m \) (for multiple strata), are clearly applicable in the case of outer joins, such as full outer join, left outer join, and right outer join, as well (as \( 0! = 1 \)). Therefore, the techniques used for equi-joins can be used for outer joins as well. As self-join involves using the same relation on both sides of the join, the equi-join allocation strategy will be applicable to it as well.

5.3.2 Non-Equi-Join

In non-equi-joins (\( \neq \)), a tuple can be joined with all non-matching tuples of other relations. The number of possible samples can be given by
\[
\prod_{j=1}^{n} \prod_{i=1}^{z} \left( C_{mm_i}^m \prod_{j=1}^{n} C_{mm_i}^m \right)
\]. Interestingly, derivation in Section 5.8.3 shows that the allocation strategy for equi-joins works in this case as well.

5.3.3 Theta Join

In theta joins, tuples are joined using the provided condition over the join columns. We provide allocation strategies, with derivation in Section 5.8.4, for theta joins for the common comparators, \(<, \leq, >, \text{ and } \geq \). We first consider the case of the \( \leq \) comparator. First, we number the strata from 1 to \( n \) in \textit{descending} order. Next, we determine approximate value of \( A \) using binary search, so that \( k = \sum_{j=1}^{n} \left( A^j \sum_{i=1}^{z} m_i^j \right) \). Then, we allocate the sample size
using proportional allocation for a stratum $j$, $k^j = A^j \times \sum_{i=1}^{z} m_i^j$. Finally, we allocate $k^j$ between the relations, $mm_i^j = k^j \times \frac{m_i^j}{\sum_{i=1}^{z} m_i^j}$.

Our derivation also shows that the only difference in the algorithm, when using the $<$ comparator instead of $\leq$ operator, is in finding $A$, such that $k = \sum_{j=1}^{n} A^{j^i} \sum_{i=1}^{z} m_i^j$. Other steps remain the same. These results can be extended for the comparators, $>$ and $\geq$, by reversing the strata order.

5.3.4 Cross Join

Cross join involves performing a cartesian product between the relations. The number of possible samples can be given by $\prod_{i=1}^{z} C_{|R_i|}^{\frac{|R_i|}{|S_i|}}$. This expression can be directly framed into the expression for maximizing randomness for a single stratum, $\prod_{i=1}^{z} C_{mm_i}^{m_i}$. The equation for space constraint can be reframed similarly as well. Hence, we use proportional allocation here, with the sample size given by $|S_i| = \frac{|R_i|}{\sum_{j=1}^{|R_j|}}$.

5.4 Non-Correlated Sampling

We have looked at techniques to maximize randomness of correlated samples of joins. Here, we look at its complementary use case and present its problem statement as follows: 

For the case where only non-correlated samples are acceptable, determine the sample allocation strategy to minimize sample size for the specified join sampling rate. This is known to be a hard problem due to sample inflation (Section 5.1.1.2). Similar to Chaudhuri et al. [31], we consider different availabilities of statistics and indexes. We look at some of the issues in their state-of-the-art algorithms, and provide enhancements to them. We then provide an algorithm for the case where statistics and indexes are available over both relations (STRATJOIN).
5.4.1 Enhancement to Group-Sample

Group-Sample is the state-of-the-art algorithm for the case of statistics being available over one of the relations. We demonstrate its shortcoming (Section 5.4.1.1), provide an algorithm that rectifies it (Section 5.4.1.2), show the theoretical proof of its correctness (Section 5.4.1.3), provide its time (Section 5.4.1.4) and space (Section 5.4.1.5) complexities, and discuss a major enhancement if sorting were possible (Section 5.4.1.6).

5.4.1.1 Issues in Group-Sample

We briefly describe Group-Sample – please refer [31] for details. Using the statistics over $R_2$, Group-Sample samples $R_1$ in a streaming fashion, by weighting each tuple in $R_1$ by the number of tuples in $R_2$ that can join with it, resulting in $S_1$. Next, it joins $S_1$ with $R_2$, generating a group of size $m_2(t_1.A)$ for each sampled tuple $t_1$ in $S_1$. Finally, it chooses a single tuple from every group in a streaming fashion.

Joining $S_1$ with $R_2$ will result in the intermediate materialized data having a size of $f \times \sum_{a \in \text{Strata}} m_1(a) \times m_2(a) \times m_2(a)$ on average. This can result in the intermediate data size exceeding the size of join between non-sampled relations $- \sum_{a \in \text{Strata}} m_1(a) \times m_2(a)$ – which renders sampling counter-productive. Further, a scan is then needed over the intermediate materialized data to choose random tuples from each group, at a time complexity of $O(f \times \sum_{a \in \text{Strata}} m_1(a) \times m_2(a) \times m_2(a))$.

5.4.1.2 Group-Sample-Enhanced

Group-Sample-Enhanced eliminates the need to materialize large intermediate data and scan it. Initially, it obtains a with-replacement sample $S_1$ of $R_1$, in a streaming fashion, by weighting a tuple $t$ by $m_2(t.A)$. Next, while tuples of $S_1$ are streaming by, we start scanning $R_2$ and set a counter $i$ to 0. Upon finding a tuple $t_2$ that can join with the
current tuple \( t_1 \) in \( S_1 \), we try to join \( t_1 \) with \( t_2 \) using \( \text{Bernoulli}\left(\frac{1}{m_2(t_1.A) - i + 1}\right) \). If the join is successful, we set \( i \) to 0 and continue with the next tuple in \( S_1 \).

### 5.4.1.3 Proof of Correctness

We show that the probability of \( t_1 \) joining with any of the \( m_2(t_1.A) \) tuples equals \( \frac{1}{m_2(t_1.A)} \).

Let the \( i^{th} \) tuple amongst \( m_2(t_1.A) \) tuples of \( R_2 \) be denoted by \( m_2(t_1.A)[i] \). The probability of joining it with \( t_1 \) will be

\[
P(\text{reject } m_2(t_1.A)[1]) \times P(\text{reject } m_2(t_1.A)[2]) \cdots \times P(\text{reject } m_2(t_1.A)[i-1]) \times P(\text{accept } m_2(t_1.A)[i])
\]

\[
= \left(1 - \frac{1}{m_2(t_1.A)}\right) \times \left(1 - \frac{1}{m_2(t_1.A) - 1}\right) \cdots \times \left(1 - \frac{1}{m_2(t_1.A) + 1 - (i-1)}\right) \times \frac{1}{m_2(t_1.A) - i + 1}
\]

\[
= \frac{m_2(t_1.A) - 1}{m_2(t_1.A)} \times \frac{m_2(t_1.A) - 2}{m_2(t_1.A) - 1} \cdots \times \frac{m_2(t_1.A) - i + 1}{m_2(t_1.A) - i + 2}
\times \frac{1}{m_2(t_1.A) - i + 1} = \frac{1}{m_2(t_1.A)}
\]

### 5.4.1.4 Time Complexity

As the initial sampling step is identical in both \textit{Group-Sample} and \textit{Group-Sample-Enhanced}, with a time complexity of \( f \times \sum_{a \in \text{Strata}} m_1(a) \times m_2(a) \), we provide the average time complexity of our join step. First, we look at the probability of the join happening by the \( i^{th} \) tuple.

\[
P\left( \text{join does not occur by } m_2(t_1.A)[i^{th} \text{ tuple}] \right)
\]

\[
= P(\text{reject } m_2(t_1.A)[1]) \times \cdots \times P(\text{reject } m_2(t_1.A)[i])
\]

\[
= \left(1 - \frac{1}{m_2(t_1.A)}\right) \times \left(1 - \frac{1}{m_2(t_1.A) - 1}\right) \cdots \times \left(1 - \frac{1}{m_2(t_1.A) - i + 1}\right)
\]

\[
= \frac{m_2(t_1.A) - 1}{m_2(t_1.A)} \times \frac{m_2(t_1.A) - 2}{m_2(t_1.A) - 1} \cdots \times \frac{m_2(t_1.A) - i}{m_2(t_1.A) - i + 1}
\]

\[
= \frac{m_2(t_1.A) - i}{m_2(t_1.A)}
\]

97
\[ P \left( \text{join occurs by } m_2(t_1.A) \right| i^{th \text{ tuple}} \right) = 1 - P \left( \text{join does not occur by } m_2(t_1.A) \right| i^{th \text{ tuple}} \right) = \frac{i}{m_2(t_1.A)} \]

The probability of a tuple joining by the halfway stage will be \( \frac{1}{2} \), as \( i = \frac{m_2(t_1.A)}{2} \). Thus, on average, half the tuples of a stratum will be accessed before the join occurs. As a result, assuming tuples are present in a random order in \( R_2 \), on average, half the tuples in \( R_2 \) will be accessed for joining with every tuple from \( S_1 \).

### 5.4.1.5 Space Complexity

Joining a tuple from \( S_1 \) with a random tuple from its corresponding stratum in \( R_2 \) does not materialize any intermediate data. Hence, in addition to the space required for \( S_1 \) and the output, \( f \times \sum_{a \in \text{strata}} m_1(a) \times m_2(a) \), we will only need to keep a count of the number of tuples belonging to the stratum that have been accessed so far in the join step.

### 5.4.1.6 Effect of Sorted \( R_2 \)

In the case that \( R_2 \) is not sorted, the time complexity of joining \( S_1 \) with \( R_2 \) in Group-Sample will be \( O \left( f \times (\sum_{a \in \text{strata}} m_1(a) \times m_2(a)) \times |R_2| \right) \). While Group-Sample-Enhanced improves upon it, it still needs to perform a scan accessing half the tuples of \( R_2 \) to join with every tuple from \( S_1 \). If it were possible to sort \( R_2 \), the time complexities of Group-Sample-Enhanced will be greatly reduced – the number of tuples from \( R_2 \) that need to be accessed during the join step reducing from \( O(|R_2|) \) to \( O(\log(|R_2|) + m_2(t_1.a)) \) for stratum \( a \).

### 5.4.2 Enhancement to Frequency-Partition-Sample

Frequency-Partition-Sample is applicable for the case where statistics are available for the larger strata of a single relation. It uses Group-Sample for such strata. The strata lacking
in statistics are first joined and then the output sampled. Space and time complexities are dominated by the larger strata. The shortcomings of Group-Sample will affect Frequency-Partition-Sample as well – the intermediate data can be expected to be large since Group-Sample is applied over the larger strata. Hence, our Frequency-Partition-Sample algorithm uses Group-Sample-Enhanced instead of Group-Sample for larger strata, and follows Frequency-Partition-Sample’s approach for smaller strata.

5.4.3 Enhancement to Stream-Sample

Stream-Sample is designed for the case where one of the relations, \( R_2 \), has access to indexes and statistics. In Stream-Sample, first, a with-replacement random sample, \( S_1 \), is constructed over \( R_1 \) by setting weight of a tuple \( t_1 \) to \( m_2(t_1.A) \). Next, as the tuples of \( S_1 \) are streaming by, a tuple \( t_1 \) is joined with one of the random tuples from \( m_2(t_1.A) \).

If \(|S_1|\) is materialized, it will have a size \( f \times |R_1 \bowtie R_2| \) on average, which can be larger than \(|R_1|\), rendering sampling counter-productive. This problem occurs at the stratum level – sampling rate of a stratum \( a \) in \( R_1 \) will be \( f_1(a) = \frac{f \times m_1(a) \times m_2(a)}{m_1(a)} = f \times m_2(a) \). Clearly, whenever \( f_1(a) > 1 \), sampling will be counter-productive – it will be prudent to not sample such strata, and only sample if it reduces the stratum size. Note that such an approach can be used to reduce the sample size in Group-Sample-Enhanced as well.

5.4.4 STRATJOIN – Sampling Both Relations

We now describe how to sample joins for the case where indexes and statistics are available over both relations. A simple random sample of the desired size is generated for each stratum, resulting in the output being a stratified random sample.
In the initial sampling step, for every stratum of both relations, we create a with-replacement sample of size \( f \times m_1(a) \times m_2(a) \) if \( f \times m_{\text{other}}(a) \leq 1 \) for a stratum \( a \). Otherwise, we use the entire stratum in the sample. In the next step (join), for every stratum, depending on whether none of the relations, one of the relations, or both the relations are sampled, we join the samples as explained below. If both strata are sampled, we randomly choose and join sampled tuples. We use a sampled tuple only once. Otherwise, if a single relation is sampled, while tuples from the sampled stratum are streaming by, we join them with a randomly chosen tuple from the non-sampled stratum. Otherwise, if neither relation is sampled, we join random tuples with-replacement from both strata till sampling rate is met.

### 5.5 Experiments & Evaluation

Our experiments were implemented in Java 8, and were run on an Ubuntu Linux 14.04.1 LTS system with a 24-core 2.4GHz Intel Xeon CPU, 256GB DDR3 @ 1866 MHz memory, and a 500GB @ 7200 RPM disk. Section 5.5.1 studies effectiveness of our techniques at maximizing the number of samples. Section 5.5.2 compares randomness, and error in the presence of correlation between join and measure columns, between different join techniques. Our experiments use multiple datasets to illustrate the corresponding use cases well. Their size might be limited in some cases to determine the non-sampled ground truth to compare against.

#### 5.5.1 Allocation Error

This section looks at the effectiveness of our sample allocation techniques at maximizing the number of samples in equi-joins. We validate our allocation strategy for both single stratum and multiple strata being involved in the join. To evaluate our accuracy, we find
the best solution by searching through all possible allocations. This is computationally expensive and restricts size of datasets. We used the following metrics: mean squared error – $\frac{\sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2}{n}$; mean squared relative error – $\frac{\sum_{i=1}^{n} \left( \frac{Y_i - \hat{Y}_i}{Y_i} \right)^2}{n}$; and maximum difference – $\text{Max}\left( \left\{ \left| Y_i - \hat{Y}_i \right| : i \in [1,n] \right\} \right)$; where $Y_i$ is the value for optimal allocation found using brute-force search, $\hat{Y}_i$ is our predicted value, and $n$ represents the number of observations.

5.5.1.1 Single Stratum Partitioning

Since our approach might not result in the optimal allocation (Section 5.2.5), we determine its accuracy by trying different relation and strata sizes, using 3 relations. The total population size, $|R_1| + |R_2| + |R_3|$, was varied from 150 to 1000. For each population size, all possible assignments to $|R_1|$, $|R_2|$, and $|R_3|$ were tried. We ensured that each relation had a minimum of 5 tuples. The sample size was varied from 15 to 100. We found the optimal allocation through brute-force search – searching for larger population and sample sizes was prohibitively expensive. Table 5.3 shows that all error metrics are low – validating our effectiveness in practice for the single stratum case.

<table>
<thead>
<tr>
<th>Population</th>
<th>MSE</th>
<th>MSRE</th>
<th>Max Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>0.1109</td>
<td>0.0069</td>
<td>1</td>
</tr>
<tr>
<td>200</td>
<td>0.1197</td>
<td>0.0094</td>
<td>2</td>
</tr>
<tr>
<td>300</td>
<td>0.1377</td>
<td>0.0135</td>
<td>2</td>
</tr>
<tr>
<td>400</td>
<td>0.1506</td>
<td>0.0162</td>
<td>2</td>
</tr>
<tr>
<td>500</td>
<td>0.1598</td>
<td>0.0180</td>
<td>2</td>
</tr>
<tr>
<td>600</td>
<td>0.1666</td>
<td>0.0196</td>
<td>2</td>
</tr>
<tr>
<td>700</td>
<td>0.1720</td>
<td>0.0208</td>
<td>2</td>
</tr>
<tr>
<td>800</td>
<td>0.1761</td>
<td>0.0218</td>
<td>2</td>
</tr>
<tr>
<td>900</td>
<td>0.1794</td>
<td>0.0226</td>
<td>2</td>
</tr>
<tr>
<td>1000</td>
<td>0.1821</td>
<td>0.0232</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.3: Error in Single Stratum Partitioning
5.5.1.2 Multiple Strata Partitioning

We again determined the accuracy of our allocation strategy by varying the population and sample sizes. We used 3 relations having 3 strata each, with the minimum stratum size being 3. In a similar fashion as above, the total population size was varied from 40 to 55 and the sample size was varied from 27 to 30 – again, these were the upper limits for which we could find the optimal solution through brute-force search (within a day in this case). Table 5.4 shows that all error metrics are low.

<table>
<thead>
<tr>
<th>Population</th>
<th>MSE</th>
<th>MSRE</th>
<th>Max Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>0.1439</td>
<td>0.0183</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>0.1714</td>
<td>0.0221</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>0.1097</td>
<td>0.0081</td>
<td>1</td>
</tr>
<tr>
<td>55</td>
<td>0.1087</td>
<td>0.0128</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.4: Error in Multiple Strata Partitioning

5.5.2 Comparison with Correlated Samplers

We also looked at different correlated sampling techniques in the presence of correlation between join and measure columns. Two relations with 8000 rows were generated – we have limited the data size to where a measurement could be obtained in around 10 minutes. Each relation had 2 columns – a join column and a measure column. We used two common distributions, Gaussian and Zipfian, to model the column values. We obtained similar results using the two – results using Gaussian distribution have been presented. The join columns were sampled from $\mathcal{N}(\mu = 100, \sigma = 10)$. The measure columns were sampled from $\mathcal{N}(\mu = 200, \sigma = 10)$ and $\mathcal{N}(\mu = 300, \sigma = 10)$, so that we obtain non-overlapping
values. Correlation between join and measure columns was varied from 0.1 to 0.9\(^3\). We did not find a significant change for differing correlation values – results using a correlation of 0.5 have been presented. 50 runs were performed, with different relations and samples being created in each run – the median of the measurements has been presented. We use the average relative error \(\left(\sum_{i=1}^{n} \frac{|Y_i - \hat{Y}_i|}{Y_i}\right)\), to better represent the relative error.

In our context, Ripple Join, SMS Join, and Wander Join have identical semantics, as do different cluster samplers such as End-Biased Sampling, Correlated Sampling, and Universe Sampler. In the figures, Stratified Random represents stratified random sampling of the join, which results in a non-correlated sample. Stratified Random suffers from sample inflation – it provides us with the results for the best-case scenario from the perspective of correlation.

### 5.5.2.1 Number of Possible Samples

We look at the number of possible samples as a result of different techniques (Figure 5.1), i.e. given the samples, we find \(\prod_{j=1}^{n} \prod_{i=1}^{2} C_{m_{i}}^{n_{i}}\). Note that the Y-axis is presented in loglog scale (base \(e\)) to better illustrate the growth pattern. MAXRANDJOIN consistently provides more samples than Ripple Join and Universe. STRATJOIN and Stratified Random result in the maximum possible value.

### 5.5.2.2 Sampling Error

We look at the average relative error for the measure sum (over rows) of the product of the measure columns (Figure 5.2) – other column functions that we experimented with, such as the sum of measure columns and product of measure columns, gave us similar results. Output for Universe Sampler has not provided as it resulted in larger errors – including it would

\(^3\)A column having correlation \(\rho\) with \(X\) is generated from columns \(X\) and \(Y\) by \(\rho \circ X \oplus (1 - \rho^2)^{1/2} \circ Y\). The \(\circ\) operator represents multiplication of a number with every element of an array, and \(\oplus\) operator represents the pairwise addition of two arrays.
obscure differences between the other algorithms from being seen. Ground truth for the
measure value was determined after joining the relations (without sampling). STRATIFIED
RANDOM had the least error, while MAXRANDJOIN consistently had a lower error than
Ripple Join. Interestingly, STRATJOIN had a lower error than MAXRANDJOIN for the
smallest sample size (125) indicating perhaps that avoiding correlation at the expense of
smaller output size, which STRATJOIN does, might be the better option at lower sampling
rates.

5.5.2.3 Effect of Noise on MAXRANDJOIN

We investigate MAXRANDJOIN’s resistance to common types of noise – white Gaussian
and Binomial (Figure 5.3). The standard deviation of the Gaussian noise was varied from 40
to 320. MAXRANDJOIN’s resistance starts waning with increasing standard deviation in the
Gaussian noise. It can handle Binomial error for larger sample sizes.
Figure 5.2: MAXRANDJOIN has the least error amongst correlated samplers. Interestingly, STRATJOIN has lesser error than MAXRANDJOIN for the smallest sample size.

Figure 5.3: MAXRANDJOIN is resistant to white Gaussian noise with low standard deviation. At higher sample sizes, it can handle binomial perturbations to join count information.
5.5.3 Non-correlated Sampling

We have presented extensive theoretical results describing the improvements provided by GROUP-SAMPLE-ENHANCED over Group-Sample in Section 5.4.1. In this section, we demonstrate those benefits empirically, by looking at the number of intermediate data tuples created and the time taken to obtain the join result.

5.5.3.1 Experimental Setup

We use a similar setup to that used by Chaudhuri et al. [31]. Four tables were generated with 10000 tuples each. The join column in each table had counts modeled using a Zipfian distribution. The parameter \( z \) of the Zipfian distribution was varied from 0 to 3. Four other tables with 100000 tuples were generated similarly. Each row consists of three columns – RID (integer), JoinKey (integer), and Padding (integer). We have implemented the algorithms using our custom in-memory join system. By default, we discard the first run of each experiment and report the mean of the following three runs (the runs were nearly identical for all experiments). In the figures, LHS refers to the relation with 10000 tuples while RHS refers to the one with 100000 tuples. In the legends, the numbers following the algorithm name respectively refer to the LHS and RHS skews (when available).

5.5.3.2 Intermediate Data Size

We look at the number of intermediate tuples that would be created by Group-Sample and GROUP-SAMPLE-ENHANCED for varying sampling rates (Figure 5.4) and RHS skews (Figure 5.5). They show that GROUP-SAMPLE-ENHANCED, whose intermediate data size is determined by the join size, can result in the intermediate data size being multiple orders of magnitude smaller than that of Group-Sample. Both algorithms exhibit a linear increase in the intermediate data size with increasing sampling rate. When an increase in the RHS
Figure 5.4: While an increase in the sampling rate results in a linear increase in the intermediate data size, the size required by GROUP-SAMPLE-ENHANCED is multiple orders of magnitude smaller than that of Group-Sample due to the usage of reservoir sampling-based techniques.

Figure 5.5: GROUP-SAMPLE-ENHANCED intermediate data size depends on the join size, while Group-Sample sample size does not and can increase regardless (LHS skew = 0). skew does not increase the join size (LHS skew = 0), while GROUP-SAMPLE-ENHANCED intermediate data size does not increase, that of Group-Sample does.
Figure 5.6: GROUP-SAMPLE-ENHANCED provides results around an order of magnitude sooner. It is also able to finish execution within the time limit in one of the cases where Group-Sample is not ($f = 1\%$).

Figure 5.7: GROUP-SAMPLE-ENHANCED handles increasing skew better as well.
5.5.3.3 Execution Time

We also compared execution times for the two algorithms for varying sampling rates and RHS skews (Figures 5.6 and 5.7). The time limit for a run was capped at 4 hours, which resulted in some of the results being unavailable. The execution time of GROUP-SAMPLE-EHANCED was usually an order of magnitude lower than that of Group-Sample.

5.6 Related Work

As performing joins can be expensive, in addition to sampling, efforts have been directed towards accelerating them through various other means such as GPUs [67], MapReduce [19, 161], multi-threading [17, 122], networked execution [104], etc. Sampling over joins has been incorporated in numerous industrial and research systems such as SQL Server, DB2, AQUA [3], Turbo-DBO [42], BlinkDB [5], and Quickr [83]. While these approaches target various layers of the database such as table scans, offline catalogs, and aggregation, there still exist several open opportunities.

In the context of non-correlated sampling, Olken et al. [113] show that it is possible to commute selection with sampling, but commuting projections and joins is harder, and sample a single relation. Chaudhuri et al. [31] provide better algorithms for joins and consider different availabilities of statistics and indexes. The AQUA system [3] obtains a simple random sample of a join in the primary key-foreign key scenario using join synopses, which TuG [136] extends to the many-to-many join scenario. While Gemulla et al. [54] aim to minimize the space overhead of join synopses, CS2 [158] extends join synopses by proposing correlated sample synopsis, which instead of storing a sample of the join, stores a sample of the correlated tuples. Bifocal sampling [50] recognizes multiplicative effect of
strata sizes and develops different sampling strategies based on strata sizes to estimate query size.

Research in obtaining non-correlated samples of joins stalled due to Chaudhuri et al. [31] showing the inherent hardness of the problem. However, considerable efforts have been expended towards obtaining correlated samples and their meaningful aggregation estimates through online aggregation, which allowed for error estimation during query execution [59, 60, 69, 120]. Jermaine et al. [72, 73] removed the dependence of ripple join algorithms on the data residing in memory to estimate error. The DBO [71] and Turbo-DBO [42] systems allowed processing of multiple relations in a scalable fashion. Nirkhiwale et al. [111] presented the sampling algebra inherent in these techniques. Our algorithm for sampling both sides of a join in the presence of indexes, STRATJOIN, is influenced by hash ripple joins [97], although they provide correlated samples. In Wander Join, Li et al. [91] improved ripple joins by sampling tuples from subsequent relations of the join which can join with the currently selected tuples. In this work, we aim to improve correlated samples by reducing correlation in samples using strategies that maximize the number of possible samples. Our concept of maximizing the number of samples in joins is influenced by the notion of sample randomness, which has been introduced by Kateb et al. [8], who use it to improve stratified reservoir sampling.

Some correlated sampling-based approaches such as End-Biased Sampling [44], Correlated Sampling [149], and Universe Sampler [83] join all tuples whose hash of the join key lies in a randomly chosen range of the hashed join domain. Tuples whose key hash does not lie in the selected domain are discarded. This results in the samples and the join being cluster samples [37]. Cluster samples can be useful when join and measure columns are not correlated – our experiments, indeed, show that such approaches have a large error in the
presence of correlation between the join and measure columns. In contrast, our approach provides representatives from all join keys and performs well in the presence of correlation.

### 5.7 Conclusion & Future Work

Join sampling is an interesting and important area of research. We have presented techniques to sample joins, in the context of correlated and non-correlated sampling – illustrating the benefits and drawbacks in doing so. We provided novel techniques for increasing randomness of joins when correlated samples are acceptable. We showed that our techniques to maximize join randomness were effective over varying population and sample sizes. While other correlated samplers are applicable only in the case of aggregate, approximable queries, our techniques are application-agnostic – they still result in the sample having a lower aggregate measure error when compared with other state-of-the-art correlated samplers. It affirms our intuition that increasing the number of possible samples reduces the correlation in the samples. In the context of non-correlated sampling, we provided major improvements to the state-of-the-art algorithms. We also provided an algorithm for sampling both sides of a join in the presence of statistics and indexes over both. Finally, while our algorithms use statistics over join columns, this dependency cannot be done away with for high-performance querying. While one approach for gathering such statistics has been to run regularly scheduled cron jobs, another recent one has been to collect statistics over the first pass over the data and use them over the subsequent passes [26, 83].

Going forward, several interesting avenues of research are enabled by our findings. We wish to investigate the applicability of search-based solutions to obtain theoretically perfect allocation [155]. We would like to develop an estimation framework for computing variance for our correlated sampling-based approaches [78]. We wish our work influences
others to take correlation into consideration while designing algorithms. Finally, we hope other research areas that use correlated samples, in both computer science and statistics communities, use strategies to maximize sample randomness.

5.8 Derivations

This section provides the mathematical derivations that form the basis of our allocation strategies. The notation given in Table 5.1 is used in this section – superscript and subscript denote the stratum index and the relation index, respectively.

5.8.1 Maximizing Randomness for Single Stratum Equi-Join

In this case, superscript has been omitted, since a single stratum is considered. The number of possible samples can be given by

\[ f(mm_1, \ldots, mm_z) = \prod_{i=1}^{z} \frac{m_i!}{mm_i!(m_i-mm_i)!} \]  \hspace{1cm} (5.4)

The fixed sample size constraint can be given by

\[ g(mm_1, mm_2, \ldots, mm_z) = \sum_{i=1}^{z} mm_i = k^l \]  \hspace{1cm} (5.5)

\( g \) denotes the constraint function. Lagrange multipliers gives a system of \( z \) equations, with \( i^{th} \) (\( i \in [1, z] \)) equation obtained by taking partial derivative of equation 5.4 with respect to \( mm_i \)

\[ f_{mm_i} = \left( \prod_{j=1, j \neq i}^{z} \frac{m_j!}{mm_j!(m_j-mm_j)!} \right) \times m_i! \times \frac{\partial}{\partial mm_i} \left( \frac{1}{mm_i!(m_i-mm_i)!} \right) \]  \hspace{1cm} (5.6)

Using the basic Lagrange multipliers equation of \( f_{mm_i} = \lambda \times g_{mm_i} \) and equation 5.5, we also get

\[ f_{mm_i} = \lambda \times g_{mm_i} = \lambda \times \frac{\partial \left( \sum_{i=1}^{z} mm_i \right)}{\partial mm_i} = \lambda \]  \hspace{1cm} (5.7)
Using the Gamma function to represent factorial, we get

\[(n!)' = \Gamma'(n + 1) = n! \times \left( -\gamma + \sum_{a=1}^{n} \frac{1}{a} \right) \quad (5.8)\]

where \(\gamma\) is Euler-Mascheroni constant. \(\frac{\partial}{\partial n} \left( \frac{1}{n! \times (m_i - mm_i)!} \right)\) can be simplified as

\[- \left( \frac{mm_i! \times \frac{\partial}{\partial n} ((m_i - mm_i)!) + (m_i - mm_i)! \times \frac{\partial}{\partial n} (mm_i!)}{(mm_i! \times (m_i - mm_i)!)^2} \right)\]

\[- \left( \frac{mm_i! \times (m_i - mm_i)! \times (-\gamma + \sum_{a=1}^{mm_i} \frac{1}{a}) \times (-1)}{(mm_i! \times (m_i - mm_i)!)^2} \right)\]

\[- \left( \frac{mm_i! \times mm_i! \times (-\gamma + \sum_{a=1}^{mm_i} \frac{1}{a})}{(mm_i! \times (m_i - mm_i)!)^2} \right)\]

\[- \left( \frac{mm_i! \times (m_i - mm_i)! \times (-\sum_{a=1}^{mm_i} \frac{1}{a} + \sum_{a=1}^{mm_i} \frac{1}{a})}{mm_i! \times (m_i - mm_i)!} \right)\]

\[- \left( \frac{\frac{1}{a} - \sum_{a=1}^{mm_i} \frac{1}{a}}{mm_i! \times (m_i - mm_i)!} \right)\]

Plugging this into equation 5.6 and using equation 5.7, we get

\[\lambda = \left( \prod_{j=1 \& j \neq i}^{mm_i} \frac{m_j!}{mm_j! \times (m_j - mm_j)!} \right) \times m_i! \times \frac{-\left( \sum_{a=1}^{mm_i} \frac{1}{a} - \sum_{a=1}^{mm_i} \frac{1}{a} \right)}{mm_i! \times (m_i - mm_i)!}\]

\[- \left( \prod_{j=1}^{mm_i} \frac{m_j!}{mm_j! \times (m_j - mm_j)!} \right) \times \left( \sum_{a=1}^{mm_i} \frac{1}{a} - \sum_{a=1}^{mm_i} \frac{1}{a} \right)\]

The sum of harmonic series can be approximated by

\[\sum_{a=1}^{mm_i} \frac{1}{a} = \ln(mm_i) + \gamma + \frac{1}{2 \times mm_i} - \frac{1}{12 \times mm_i^2}\]

\[\approx \ln(mm_i) + \gamma \quad (5.10)\]

for large \(mm_i\), where \(\gamma\) is the Euler-Mascheroni constant. Using the above two equations, we get

\[- \lambda = \prod_{j=1}^{mm_i} \frac{m_j!}{mm_j! \times (m_j - mm_j)!} \times (\ln(mm_i) - \ln(m_i - mm_i))\]

\[= \prod_{j=1}^{mm_i} \frac{m_j!}{mm_j! \times (m_j - mm_j)!} \times \ln \left( \frac{mm_i}{m_i - mm_i} \right) \quad (5.11)\]
Note that the term \( \left( \prod_{j=1}^{z} \frac{m_{j}!}{mm_{j}!} \right) \) is present in all \( i \in [1, z] \) equations – we denote it by \( C \), which gives us

\[
-\frac{\lambda}{C} = \ln \left( \frac{mm_i}{m_i - mm_i} \right)
\]  

(5.12)

Consider the equation for another relation \( p \)

\[
-\frac{\lambda}{C} = \ln \left( \frac{mm_p}{m_p - mm_p} \right)
\]  

(5.13)

From the above two equations, we get

\[
\frac{mm_i}{m_i - mm_i} = \frac{mm_p}{m_p - mm_p} = A
\]  

(5.14)

for a constant \( A \). Therefore,

\[
mm_i = \frac{A \times m_i}{1 + A}
\]  

(5.15)

Using equations 5.5 and 5.15, we get

\[
k^1 = \sum_{i=1}^{z} mm_i = \sum_{i=1}^{z} \frac{A \times m_i}{1 + A} = \frac{A}{1 + A} \sum_{i=1}^{z} m_i
\]

Using above two equations, we get

\[
mm_i = \left( \frac{A}{1 + A} \right) \times m_i = \frac{k^1 \times m_i}{\sum_{a=1}^{z} m_a}
\]  

(5.16)

### 5.8.2 Maximizing Randomness for Multiple Strata Equi-Join

The constraint of fixed sample size for equi-joins with multiple strata can be given as follows, given \( k^j = \sum_{i=1}^{z} mm_i^j \).

\[
k = \sum_{j=1}^{n} k^j = g(k^1, k^2 \dots k^n)
\]  

(5.17)
The number of possible samples can be given by

\[ f (k^1, k^2 \ldots k^n) = \prod_{j=1}^{n} \prod_{i=1}^{z} C_{mm_i}^{m_j} \]

\[ = \prod_{j=1}^{n} \prod_{i=1}^{z} \frac{m_j^!}{mm_i^! \times (m_j^ - mm_i^)!} \]

\[ = \prod_{j=1}^{n} \prod_{i=1}^{z-1} \frac{m_j^!}{mm_i^! \times (m_j^ - mm_i^)!} \times \frac{m_i^!}{mm_i^ \times (m_i^ - mm_i^)!} \times \frac{m_j^!}{mm_j^ \times (m_j^ - mm_j^)!} \]

\[ = \prod_{j=1}^{n} \prod_{i=1}^{z-1} \frac{m_i^!}{mm_i^ \times (m_i^ - mm_i^)!} \times \frac{m_j^!}{mm_j^ \times (m_j^ - mm_j^)!} \times \frac{m_i^!}{mm_i^ \times (m_i^ - mm_i^)!} \]

Equation 5.16 provides optimal allocation amongst relations given total allocation to a stratum. Using it in the above equation results in the number of samples increasing or staying the same. Hence, \( k^j \) \((j \in [1, n])\) are the variables in this case. Lagrange multipliers gives us the \( j^{th} \) equation as

\[ f_{kj} = \prod_{j=1}^{n} \prod_{i=1}^{z-1} \frac{m_i^!}{mm_i^ \times (m_i^ - mm_i^)!} \times m_i^! \times \frac{1}{mm_j^ \times (m_j^ - mm_j^)!} \times \frac{1}{mm_i^ \times (m_i^ - mm_i^)!} \]

\[ \frac{\partial}{\partial k^j} \left( \frac{1}{(k^j - \sum_{i=1}^{z-1} mm_i^)! \times (m_i^ - (k^j - \sum_{i=1}^{z-1} mm_i^)!)!} \right) \]

We simplify \[ \frac{\partial}{\partial k^j} \left( \frac{1}{(k^j - \sum_{i=1}^{z-1} mm_i^)! \times (m_i^ - (k^j - \sum_{i=1}^{z-1} mm_i^)!)!} \right) \]

as,

\[ \frac{-1}{(k^j - \sum_{i=1}^{z-1} mm_i^)! \times (m_i^ - (k^j - \sum_{i=1}^{z-1} mm_i^)!)!} \times m_i^! \times (k^j - \sum_{i=1}^{z-1} mm_i^)! \times \left( -\gamma + \sum_{l=1}^{k^j-\sum_{i=1}^{z-1} mm_i^} \frac{1}{a} \right) \times \left( m_i^ - (k^j - \sum_{i=1}^{z-1} mm_i^) \right) \]

\[ \left( k^j - \sum_{i=1}^{z-1} mm_i^ \right) \times \left( m_i^ - (k^j - \sum_{i=1}^{z-1} mm_i^) \right) \times \left( -\gamma + \sum_{l=1}^{k^j-\sum_{i=1}^{z-1} mm_i^} \frac{1}{a} \right) \]

(5.20)
Consider the equation for another stratum $p$

$$p = \lambda$$

From the above 2 equations, given a constant $A$, we get

$$k^j - \sum_{i=1}^{z-1} mm_i^j = k^p - \sum_{i=1}^{z-1} mm_i^p = A$$  \hspace{1cm} (5.23)
Using equation 5.16 and the above equation, we get

\[
A = \frac{k^j - \sum_{i=1}^{z-1} k^j \times m^j_i}{\sum_{a=1}^{n} m^j_a - \sum_{i=1}^{z-1} m^j_i} = \frac{k^j \left( 1 - \frac{\sum_{i=1}^{z-1} m^j_i}{\sum_{a=1}^{n} m^j_a} \right)}{\sum_{a=1}^{z} m^j_a}
\]  

(5.24)

\[
k^j = A \times \sum_{a=1}^{z} m^j_a
\]  

(5.25)

Using above equation and equation 5.17, we get

\[
k = \sum_{j=1}^{n} \left( A \times \sum_{a=1}^{z} m^j_a \right)
\]  

(5.26)

\[
A = \frac{k}{\sum_{j=1}^{n} \sum_{a=1}^{z} m^j_a}
\]  

(5.27)

Using equations 5.25 and 5.27, we get

\[
k^j = \frac{k \times \sum_{i=1}^{z} m^j_i}{\sum_{j=1}^{n} \sum_{i=1}^{z} m^j_i}
\]  

(5.28)

### 5.8.3 Maximizing Randomness For Non-Equi-Joins

Number of samples for non-equi-joins can be given by

\[
\prod_{j=1}^{n} \prod_{i=1}^{z} \left( \frac{C_{mn j}^{m^j_i} \prod_{j=1}^{i=1} \prod_{i=1}^{z} C_{m n i}^{m^j_i}}{C_{m n i}^{m^j_i} \prod_{a=1}^{n} C_{m n a}^{m^i_a} \prod_{b=1}^{z} C_{m n b}^{m^b_b}} \right)
\]

\[
= \prod_{j=1}^{n} \prod_{i=1}^{z} \left( \frac{C_{mn j}^{m^j_i} \prod_{j=1}^{i=1} \prod_{i=1}^{z} C_{m n i}^{m^j_i}}{C_{m n i}^{m^j_i} \prod_{a=1}^{n} C_{m n a}^{m^i_a} \prod_{b=1}^{z} C_{m n b}^{m^b_b}} \right)
\]

\[
= \prod_{j=1}^{n} \prod_{i=1}^{z} \left( \frac{C_{mn j}^{m^j_i} K}{C_{m n a}^{m^i_a} \prod_{a=1}^{n} C_{m n a}^{m^i_a} \prod_{b=1}^{z} C_{m n b}^{m^b_b}} \right)
\]
\[ K^{n z} \prod_{j=1}^{n} \prod_{i=1}^{z} \left( \frac{C_{m_i}^{m_j}}{\prod_{a=1}^{n} C_{m_i}^{m_j} \prod_{b=1}^{z} C_{m_i}^{m_i}} \right) \]

\[ \cdots \text{where } K = \prod_{j=1}^{n} \prod_{i=1}^{z} C_{m_i}^{m_j} \]

\[ = K^{n z} \prod_{j=1}^{n} \prod_{i=1}^{z} C_{m_i}^{m_j} \]

\[ = K^{n z} \prod_{j=1}^{n} \prod_{i=1}^{z} \left( \frac{1}{\prod_{a=1}^{n} C_{m_i}^{m_j} \prod_{b=1}^{z} C_{m_i}^{m_i}} \right) \]

\[ = K^{n z} K^2 \frac{1}{\prod_{j=1}^{n} K} \frac{1}{\prod_{i=1}^{z} K} = K^{n z} K^2 \frac{1}{K^n} \frac{1}{K^z} = K^{n z + 2n - z} \]

\[ K \text{ can be maximized using proportional allocation (Section 5.8.2). Hence, we use the equi-join allocation strategy here as well.} \]

### 5.8.4 Maximizing Randomness For Theta Joins

For \( \leq \) comparator, the number of samples can be given by

\[ \Pi_{i=1}^{n} \Pi_{j=1}^{n} C_{m_i}^{m_j} \Pi_{i \neq i}^{n} \Pi_{j=1}^{j} C_{m_i}^{m_i} \text{ and its log by} \]

\[ \sum_{i=1}^{z} \sum_{j=1}^{n} \left( \ln \left( C_{m_i}^{m_i} \right) + \sum_{i=1}^{z} \sum_{ii \neq i}^{j} \ln \left( C_{m_i}^{m_i} \right) \right) \]

\[ = \sum_{i=1}^{z} \sum_{j=1}^{n} \ln \left( C_{m_i}^{m_i} \right) + \sum_{i=1}^{z} \sum_{ii \neq i}^{j} \ln \left( C_{m_i}^{m_i} \right) \]

\[ = \sum_{i=1}^{z} \sum_{j=1}^{n} \ln \left( C_{m_i}^{m_i} \right) + (z - 1) \times \sum_{i=1}^{z} \sum_{j=1}^{n} \sum_{j=1}^{j} \ln \left( C_{m_i}^{m_i} \right) \]

\[ = \sum_{i=1}^{z} \sum_{j=1}^{n} \ln \left( C_{m_i}^{m_i} \right) + (z - 1) \times \sum_{i=1}^{z} \sum_{j=1}^{n} (n - j + 1) \ln \left( C_{m_i}^{m_i} \right) \]
\[
(z - 1) \times \sum_{i=1}^{\bar{z}} \sum_{j=1}^{n} (n - j + 1 + \frac{1}{z - 1}) \times \ln \left( \frac{m_i^j}{mm_i^j} \right)
\]

\approx (z - 1) \times \sum_{i=1}^{\bar{z}} \sum_{j=1}^{n} (n - j + 1) \times \ln \left( \frac{m_i^j}{mm_i^j} \right)

= (z - 1) \times \sum_{i=1}^{\bar{z}} \sum_{j=1}^{n} j \times \ln \left( \frac{m_i^j}{mm_i^j} \right)

Strata order was reversed in the last step. We now have to minimize \( \sum_{i=1}^{\bar{z}} \sum_{j=1}^{n} j \times (\ln(mm_i^j) + \ln((m_i^j - mm_i^j)!)) \), with \( k = \sum_{i=1}^{\bar{z}} \sum_{j=1}^{n} mm_i^j = g(mm_i^1...mm_i^n) \). Lagrange multipliers gives us the \( \{i, j\}^{th} (i \in [1, z], j \in [1, n]) \) equation as

\[
f_{mm_i^j} = \lambda \frac{\partial g}{\partial mm_i^j} = \lambda = \frac{\partial}{\partial mm_i^j} (j(\ln(mm_i^j) + \ln((m_i^j - mm_i^j)!)))
\]

\[
= j \left( \frac{1}{mm_i^j!} \right) \left( - \gamma + \sum_{a=1}^{mm_i^j} \frac{1}{a} \right) +
\]

\[
j \left( \frac{1}{(m_i^j - mm_i^j)!} \right) \left( m_i^j - mm_i^j \right) ! \left( - \gamma + \sum_{a=1}^{m_i^j - mm_i^j} \frac{1}{a} \right) (-1)
\]

\[
= j \left( \sum_{a=1}^{mm_i^j} \frac{1}{a} - \sum_{a=1}^{m_i^j - mm_i^j} \frac{1}{a} \right)
\]

\[
= j(\ln(mm_i^j) + \gamma - \ln(m_i^j - mm_i^j) - \gamma) = j \left( \ln \left( \frac{mm_i^j}{m_i^j - mm_i^j} \right) \right)
\]

\[
= - j \left( \ln \left( \frac{m_i^j}{mm_i^j} \right) - 1 \right) \approx - j \left( \ln \left( \frac{m_i^j}{mm_i^j} \right) \right)
\]

For every equation, \( \left( \frac{m_i^j}{mm_i^j} \right)^j \) is constant. For any stratum as \( j \) is constant, \( m_i^j / mm_i^j \) will be constant – showing that proportional allocation should be used to allocate samples for a stratum between relations. Therefore, \( mm_i^j = k_i^j m_i^j / \sum_{i=1}^{z} m_i^j \). Therefore, for all strata, using above two equations, \( \left( k_i^j / \sum_{i=1}^{z} m_i^j \right)^j \) is constant, denoted by \( A \). Therefore, \( k_i^j = A_i^j \sum_{i=1}^{z} m_i^j \) and \( k = \sum_{j=1}^{n} k_j^j = \sum_{j=1}^{n} A_i^j \sum_{i=1}^{z} m_i^j \). A is the only unknown and can be found using binary search within error. Finding \( A \) lets us find \( k_i^j \), which leads us to finding \( mm_i^j \).
Following a similar line of reasoning, the only difference in the algorithm is in determining $A$, given by $k^j = A^{1/z} \sum_{i=1}^{z} m_i^j$. 
Chapter 6: An Execution Framework for Highly Interactive Querying

6.1 Introduction

Interactive experiences using modern devices such as netbooks, iPads, smart watches, Google Glass, etc. have become increasingly popular. Constructing a query interactively is common whether it be autocompletion [74] and instant search for search engines, or using gestures on iPads and touch-based devices [106], or through image analysis in augmented reality interfaces [124, 140]. In such cases, the user’s intent (and the final query) is not explicitly clear to the system during the construction of the query. The query input terminates when the user finalizes her actions and is happy with the constructed query. User studies have found that this type of query construction occurs over query specification sessions that take in the order of tens of seconds [106], and that interactive-level latency necessitates response within the sub-second range [29, 36, 101, 123]. Thus, the interface can provide a preview of the results while the query is being formulated, thereby, assisting the user to formulate the final query. Since the traditional database paradigm is that of “query→result”, the concept of interactive-level feedback during the construction of a query is challenging. Traditional database engines are unable to effectively handle such workloads within interactive time constraints. As shown in the following sections, it is difficult to handle these tasks at the application layer and requires a rethinking of the underlying database design.
6.1.1 Interactivity & Ambiguous Query Intent

Often, the user is not completely aware of the data, the schema, or the query language. In this scenario, it becomes difficult for the user to translate her query intent into an articulate and unambiguous query, which constitutes the query specification process. Before the system goes through the “query→result” process, the user must finish the “ambiguous intent→unambiguous query” step [105]. While decades of work exists in database systems towards solving the former task, the challenge of aiding the user to articulate her intended query is comparatively novel.

Providing the user with an instantaneous/interactive feedback allows the user to understand the result space better, which in turn aids in specifying the query. For a complex database, the space of possible queries is large. Furthermore, depending on the size of the data, executing every candidate query may prove to be prohibitively expensive, thereby, hindering exploratory querying. Supporting such level of interactivity at the application layer, for example with caching, is impractical since both the number of queries and the size of the results might be infeasibly large. Thus, we can see that there is a clear need for the underlying database to provide interactive feedback for such queries. Most of these queries are ephemeral in nature – they only exist to guide the user during the query specification process and are usually useful for a few (milli)seconds with only one of them being considered finally as the user intended unambiguous query.

As highly interactive workloads become increasingly common, there exists a strong need for the databases to cope with their unique requirements. Before we expand on a formal query model in Section 6.2, we motivate and characterize our use case with a real-world example below.
6.1.2 Applications

Consider the case of performing an exploratory equijoin using hand gestures on a visual query interface running on a multitouch iPad touchscreen, similar to the one described in [106]. In this scenario, it is unclear which of the two columns of each of the tables the user wishes to join until the user approves of the system’s recognition of her intended gesture. A preview of the results enables the user to express her intent faster. This scenario is quite common for the query formulation process – real-world databases sometimes lack the documentation or primary/foreign key relationships explicitly annotated on the schema, and thus, the user has to discover these relationships using the time-consuming trial and error method. As per the query specification gesture showed in [106], let the two tables to be joined be denoted by $T_1$ and $T_2$. Then, the query at hand becomes:

$$Q_{\text{gesture}} = T_1 \bowtie T_2$$

where the conditions on the JOIN are determined by the user. During the gestural query articulation process, as the user brings the two columns of each of the tables close to each other, the system is not certain of what the final user query would be and shows a preview of the results for the most likely intended queries. Assuming the two tables $T_1$ and $T_2$ have $c_1$ and $c_2$ columns respectively, in the worst case, there can be $c_1 \times c_2$ different joins possible. Since in this scenario, the interface is a touchscreen, it encourages a high level of interactivity, and thus the rate of exploration increases – the user can consider each of the combinations in a rapid and arbitrary manner. Further exacerbating the situation is the fact that the tables might be huge, resulting in each join taking a long time to execute completely. As the user explores through all the possible combinations, she settles on one of the JOIN combinations as her final intended query.
6.1.3 Challenges

Interactive querying provides us with numerous interesting challenges as shown by the motivating examples.

- **Number of Queries:** The number of possible queries can be large. Even in the case of a single column equijoin between 2 tables $T_1$ and $T_2$ having $c_1$ and $c_2$ columns respectively, the maximum number of queries equals $c_1 \times c_2$, processing all of which can get unwieldy for large tables. The user expects interactive response times and executing each query as the user transitions between them would overwhelm any database.

- **Data Size and Query Scheduling:** The size of the data can be large enough to disallow returning the results to the user within interactive time bounds. This creates the following interesting scheduling problems. If a user is rapidly transitioning from one query to another, which query do we prioritize for execution? Given that the user is interested only in a smaller preview of the results, should we terminate the query after obtaining a few results? Further, given that the user might become disinterested in a query if the results take too long, when do we *give up* on a query and terminate its execution?

- **Query Variability:** The number of different queries produced in an interactive setting can be large. During the query specification process, the space of possible queries is initially
large, and the user may consider numerous queries before finalizing her intent. Thus, the rate of issuing (and canceling) queries to the engine poses a serious challenge.

- **Response Time vs. Throughput:** While increasing the throughput of the joins has been researched thoroughly, decreasing the response time has not been studied at the same depth. Current research primarily focuses on decreasing the overall query execution time, at the expense of responsiveness. Interactivity requires a very short response time, while throughput although important, is not the primary concern. Thus, we can see the need to provide techniques for decreasing the response time while maintaining a high throughput.

- **Highly Interactive Querying:** Interactive querying often refers to short response time, in the order of seconds and minutes. Highly Interactive Querying, on the other hand, requires 1. response time within milliseconds at the most (in this work, we assume 500ms), 2. responsiveness - the system should provide immediate feedback for each user request, 3. adequate number of results for the user to explore (we use 500 results as the threshold). Current systems are not designed to be used within such constraints, and as shown later do not respond well.

### 6.2 The Query Intent Model

We formalize our problem statement by describing the *intent space*, *query intent*, and *query intent transition*. The set of valid queries that can be issued to the database at any given time is considered as the **intent space**. Thus, the **query intent** is defined as the likelihood of each query in that space, represented by a probability distribution function (PDF) over all possible queries in the intent space. Without any input, the PDF is uniform – all queries are equally likely. At the end of the query specification, all but one of the queries
are ruled out (i.e. probability set to 0) and the probability of the unambiguous intended query is set to 1.0.

During interaction (such as the dragging gesture on the iPad), the user starts with a perfectly ambiguous query and each action results in a change in the likelihood of the queries (Section 6.3.1). We can formalize this concept by

\[ \tilde{Q} = \{ PDF_{t=1..T} (Q_{1..q(t)}(T_{1..k})) \} \quad (6.1) \]

where \( \tilde{Q} \), the query intent transition, is the set of all speculative queries during the lifetime of the specification of the query; \( T_{1..k} \) are the tables on which the interactive query acts; and \( t = 1..T \) are the points in time through the query intent transition. The PDF holds the first order Markovian property, i.e. the PDF at time \( t + 1 \) depends only on the PDF at time \( t \) and the query at hand. We can represent the query intent transition of a user in the gestural join example simplistically as:

\[ \tilde{Q} = T_1 \bowtie T_2 \quad (6.2) \]

where \( \tilde{Q} \) is the output of an equijoin between relations \( T_1 \) and \( T_2 \) and \( \bowtie \) denotes that the construction of the \texttt{JOIN} query is in flux – the PDFs of the participant columns of both \( T_1 \) and \( T_2 \) can change rapidly over time during the construction of the query. This query construction process is aided by allowing the user to peek at the results of the most likely queries.

### 6.2.1 Variability of Intent Transitions

Different interfaces and applications allow for different rates of change of the query PDF. For example, in the gestural querying example, a user may initially perform a very rapid movement while locating the right columns, and then towards the end of the transition, slow down as she finds the right set of columns and peruses the result preview more carefully.
We define this behavior as bursty, or highly variable, since the moment the user begins the interaction with the system, a burst of all the queries will be pushed to the DBMS. Thus, depending on the application use case, it is important to consider the variability of the query intent transition.

### 6.2.2 Execution Strategies

Given that the query intent transition represents the rapidly changing likelihoods of possible queries, the goal of the database is to provide useful results to the user during this process. We list two execution strategies that an interactive engine should consider.

**Full Execution:** In cases where all likely queries will need to be serviced, such as all objects on screen needing to be annotated, a tremendous load is put on the database if the number of queries is high. We consider this as *full execution*, where the database engine has to execute all queries in the query intent space, i.e., all queries with a non-zero probability in the PDF at any given time.
**Top-k Execution:** In applications such as the multitouch gestural join, it is possible to save some computing resources by using the *top-k* strategy. Since only the most likely query is previewed to the user, there is lesser value in executing unlikely queries, other than due to the fact that they might become the top query during the query intent transition. Thus, instead of using *top-1*, it may be prudent to choose *top-k*, to allow for some variability in the intent transition. Selecting the right *k* is a difficult task – a low *k* value would result in bursts of queries being sent to the engine, while a high value of *k* would behave like the full execution approach.

### 6.3 The FluxQuery Framework

In this section, we describe the framework used to solve the issues raised earlier. We detail our system architecture, strategies for sharing resources, the changes that need to be made to a regular database engine to allow handling interactive joins and how a cyclic scan can utilize these changes, and provide the necessary algorithms and protocols.

*DICE* and *Sesame* speed up queries over a single table. They do not consider the case of joins. As executing join queries is expensive, speculation techniques used by *DICE* and *Sesame* cannot be directly applied to joins. Further, including joins naively would increase the number of speculative queries multiplicatively. Our work on join sampling looks at the problem from the point of view of a single join query. It does not consider multiple queries that are generated during the query specification process as a result of interactive devices. Thus, achieving interactivity during the join specification process needs architectural changes throughout the systems, especially at the execution layer.
6.3.1 System Architecture

We propose a main-memory execution engine based on the idea of continual circular clock scans [16, 21, 146], where concurrent queries register to ongoing scans, much like passengers getting on and off a cyclic elevator. This family of scan strategies allows a reduction in latency, and prioritization of buffer page loads, and is ideal for interactive query execution. The most likely queries the user might intend to execute are registered for execution and share scans if possible. During the gestural articulation, likelihoods for the queries change, requiring registration of new queries, removal of existing ones, or changing the priorities of the already executing queries.

Figure 6.3: FluxQuery Architecture

Figure 6.3 shows the system architecture. A client builds a query and provides it to the server, which executes the queries and returns the results. Each client has an optional server process within the DB engine, which transfers the client commands to the engine.
A single server process handles requests from multiple clients. Its sole responsibility is to transfer the client commands to the appropriate join process for both query registration as well as query execution. A significant effect of this design is the ability to issue commands while the client is waiting for results of previously issued queries.

Each join process performs a cyclic join. It enables sharing of resources between different queries in order to provide the users their results as quickly as possible while minimizing the overall system overhead. In this work, we focus on join, which is a key query operator in the interactive setting. Work has been done in the past in the area of simple filtering (SharedDB [55]). Although our approach is similar, we cover joins in greater detail and provide techniques to decrease the response time by allowing queries to begin execution sooner.

In the following subsections, we present techniques to reduce the limitation on the number of queries that can be executed in parallel using three different types of joins – Cyclic Hash Join (CHJ), Cyclic Nested Loop Join (CNL), and our novel algorithms FluxJoin (FJ) and Fast FluxJoin (FFJ).

### 6.3.2 Preliminaries

#### 6.3.2.1 Shared resources and interactivity

There exist multiple ways to share resources such as latches, connections, processes, etc. We favor two techniques – Shared Servers [88] and Shared Scans [55].

As opposed to the previous work, we focus on providing results within interactive response times. In the Shared Server approach, the CPU is assumed to be the congested resource and, hence, the access to it is controlled by the Shared Server processes. To do so, the session itself is shared among different processes, however, the data is not shared between
different queries or sessions. This results in a decreased load on the system at the expense of increased response time. On the other hand, Shared Scans utilize the same table scan for multiple queries to provide results within a predictable interval. It introduces a significant delay between the query being introduced in the system and starting execution. Thus, we can see that both approaches have significant drawbacks with respect to interactivity since the start of a query is delayed significantly.

A vital challenge we face is improving the interactivity and decreasing the response time, without harming the overall query performance. We propose using shared latches within the cyclic scan at a higher granularity.

6.3.2.2 Join Types

The most commonly used join types are the Nested Loop Join (NL) and the Hash Join (HJ), which we focus on in this work. In NL, for each record of the bigger table, we iterate over each record of the smaller table and perform the join. Each tuple combination that matches the join criteria is then emitted to the client. In HJ, a hashmap is built on the smaller table using the join key. The hash value of the join key of each record of the other table is searched in the already built hashmap, returning the tuples that match the join criteria to the client.

The benefits of using a Hash Join over Nested Loop Join in some cases are apparent. The performance of a lookup for each record is improved to an average of $O(1)$ from $O(N)$ where $N$ is the number of records in the hashed table. The main disadvantage of the hash join is the consumption of additional memory. Also in the cases where a few hash buckets contain all the values, the benefits of hashing are reduced.

The choice of which join a DBMS should use is done by the optimizer engine. The decisions taken by the optimizer, such as the join type that should be used and which scan
model to use might be affected by the cyclic techniques presented here, but we do not focus on it in this work. The optimizer output, which is a query execution plan, should be handed over either to a regular execution process or to a shared join process for execution.

In the case of Hash Joins, we cache the maps after building them. These maps are reused across queries, similarly to indexes. In the traditional DBMS setting, on the other hand, even though two queries might be using hashmaps on the same columns, the maps are built separately for each. Building a large number of hashmaps might exhaust the machine’s memory and reduces the effectiveness of the database caches.

### 6.3.3 Cyclic scan

Using a cyclic scan reduces contention and improves the query response time. It also enables creating new join operators which fit the interactive scenarios better than the current ones. Our cyclic scan approach can be classified as an *always executing scan*.

In the cyclic approach, a single thread in a cyclic manner continuously scans the external table. For each block of the external table, each block of the internal table is scanned or the relevant hashmap searched. When a query is issued, it does not start execution immediately. Instead, it registers itself in a queue of future queries to run. Query modifications such as cancellations and priority changes are performed in a similar fashion. Depending on the granularity of the *safe points* (elaborated in Section 6.3.4, safe points can be briefly described as the logical points to modify the list of queries the join process operates on), we check if there are any pending queries that can start execution. If such queries exist, we start the execution of the new query from the current scan position, where its execution will also be stopped.
6.3.4 Consideration: Safe points

We define safe point to be the place within the cyclic join where we can freeze the join execution process and modify the query execution list – add a query, remove a query, or change query properties. Checking for modifications to the execution list can pose a significant overhead, especially since modifications to the list are infrequent compared to block switches. For decreasing the overhead, we enter a safe point using an optimistic approach, (without locking the objects first using a lock-free queue), to check if there are any pending modification requests.

The optimal place for the safe point is just before latching the data blocks since the safe point is entered often enough for query modifications to not delay the query execution start substantially (which is less than 1ms), while limiting the stall imposed on already running queries. In addition to not delaying query execution start time substantially, the effect of query execution list modification on the performance of already executing queries is limited. The only delay a query will experience before it starts running on the execution thread is the delay between the time it is registered and the time the execution thread enters a safe point, which is short and proportional to the number of queries the join process is executing.

6.3.5 Algorithms

We introduce the four different algorithms - CNL (Cyclic Nested Loop), CHJ (Cyclic Hash Join), FJ (the FluxJoin), and FFJ (the Fast Flux Join). FJ draws from both CNL and CHJ. It has a lower response time compared with CHJ, and a lower total execution time compared with CNL. It is based on the observation that multiple queries can share the same hashmaps built over the internal table, which avoids scans over internal table blocks for multiple queries that join using the same column. Instead of scanning over the internal table...
blocks to match rows to an external table block, we build a map over the internal block to speed up lookup for matching rows which decreases the response time.

### 6.3.5.1 Cyclic Nested Loop Join

```python
CyclicalNestedLoopJoin()
1     while True
2       do
3         for each Block EB in External Table
4             do
5                 for each Block IB in Internal Table
6                     do
7                         Queries ← SafePointAddRemoveQueries(Queries, AddQueries, RemoveQueries)
8                         for each Row ER in EB
9                           do
10                              for each Row IR in IB
11                                do
12                                   for each Query Q in Queries
13                                       do
14                                         if Q.Join(ER, IR)
15                                           then Emit Results To Clients
```

Algorithm 7: Cyclic Nested Loop Join

In Algorithm 7, we show the execution thread for CNL. We implement the cyclic scan by always executing full table scan on both external and internal tables. We execute the safe point for modifying the query execution list. Afterward, we execute the Nested Loop Join.

### 6.3.5.2 Cyclic Hash Join

In Algorithm 8, we show the execution thread for CHJ. In CHJ, the internal table blocks are not accessed directly as in CNL – the rows are accessed through the hashmaps. Therefore, it does not iterate over the internal table. The rest of the algorithm is similar to that of Algorithm 7, whereas the join implementation is that of the Hash Join.
```c
CyclicHashJoin()
while True
for each Block EB in External Table
do
Queries ← SAFEPOINTADDREMOVEQUERIES(Queries, AddQueries, RemoveQueries)
for each Row ER in EB
do
for each Query Q in Queries
do
MAP ← CurrentMaps.GET(Q.HASHMAPCOL)
if Q.JOIN(ER, MAP)
then Emit Results To Clients
```

Algorithm 8: Cyclic Hash Join

### 6.3.5.3 FJ - The FluxJoin

In this section, we introduce the FluxJoin. It is designed to be used in a cyclic fashion. FJ is based on the observation that the Nested Loop Join is extremely responsive, but slow, while the Hash Join is fast, but slower to respond since the initial hashmap creation results in a scan of the entire inner table. We developed a combination of the two algorithms, Nested Loop Join and Hash Join, that allows the usage of hashmaps for accelerating the Nested Loop Join. This was done by changing the granularity of the hashmap from being on the full table to being on a block.

We reuse the block-hashmaps only for each external-internal block combinations and do not cache them for later use as the case in CHJ. FJ uses limited memory, and therefore, is able to handle the cases where CHJ cannot fit all the hashmaps in memory.

In FJ, as seen in Algorithm 9, we execute a modified CNL, in which in the innermost iteration, we do not iterate on each row within the block. Instead, we build hashmaps on the current inner block for each column that is queried and search it using the external block rows. We reuse the same hashmaps for all the queries, as done in the CHJ algorithm. Maps are held only for the duration of the internal iteration.
Algorithm 9: FluxJoin

Thus, we can see that FJ is the optimal join to use in the interactive setting. The amount of overhead memory it consumes is negligible and fits well within the different levels of system caches. The performance of FJ is similar to that of CNL when a single query is executed. Yet, when multiple queries are executed on the same column, the performance improvement would improve linearly with regards to the number of executing queries due to sharing of the maps. FJ cannot perform as well as HJ due to the repeated creation of the hashmaps.

6.3.5.4 FFJ - Fast Flux-Join

In our current FJ algorithm, we do not reuse the hashmaps across different external blocks, and therefore the performance is closer to that of CNL than that of CHJ. In the case where all hashmaps can fit in memory, we can build an additional full hashmap for each column, while the FJ emits results for each block – preserving its CNL-like low response time. Later external blocks can use the fully built hashmap, resulting in the performance advantages of CHJ.
6.3.6 Sampling & Cyclic Joins

There are multiple aspects of sampling that can be used in the context of interactive query execution. It is clear that a row sampled while keeping a particular join combination in consideration is usually not useful for another join combination. It is also possible to sample rows that are likely to join over multiple columns so as to provide the user with consistent data view over multiple joins, thus, trading off randomness over consistency. FJ and FFJ can be seen to use non-random online samples of the data to achieve interactivity. Obtaining random samples over joins is a hard problem as we will see in the next chapter.

6.4 Experiments & Evaluation

We evaluate our system’s scalability and response time, i.e. interactivity. We measure our system’s performance over increasing datasets (table size of between 10KB and 100GB) and increasing number of queries (between 1 to 2048) while determining whether each experiment met the interactivity constraints. We demonstrate the scalability of our system – all our algorithms successfully process thousands of simultaneous queries, while FJ and FFJ provide results within interactive response time.

6.4.1 Experimental Setup

We performed our experiments on machines with the following configurations:

MACHINE244GB : RedHat (kernel 2.6), Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz x 4 (32 cores), 20 MB L2 cache with 244 GB of RAM (EC2 r3.8xlarge).

MACHINE4GB : CentOS (kernel 2.6), Intel(R) Xeon(R) CPU X3210 @ 2.13GHz x 1 (4 cores), 8 MB L2 cache with 4 GB of RAM.
6.4.1.1 Datasets

Since a benchmark for interactivity for large datasets does not exist, we synthesized the data. Unless reported otherwise, the number of columns in each table is set to 16. Between table sizes of 10 KB to 100 GB, our experiments cover the entire spectrum of expected requirements from in-memory databases.

6.4.1.2 Workloads

The workloads determine how many queries will be executed in parallel and when each begins and ends execution. We constructed 2 workloads based on different burstiness behaviors (Section 6.2.2) of joins – WORKLOADFULL (default) and WORKLOADTOPK. We also experimented with a real workload, provided by the authors of [106], WORKLOADREAL. In this workload, users gesturally interact with their system to build simple queries.

6.4.1.3 Baseline DBMS

We compared our system with System X, a commercial DBMS system; System Y, a commercial vector-based DBMS system; and PostgreSQL 9.3. We focused on these systems since they are used in data warehouses and represent research as well as industry.

Although System Y outperformed System X for queries involving a single relation, System X outperformed System Y in every experiment involving joins. Hence, we have not reported the performance of System Y.

We were not able to obtain either implementation, source code, or executables of research DBMS engines more closely related to ours despite multiple attempts to contact authors of multiple papers. Hence, we discuss the design principles for all related work in the absence of these specific implementations.
6.4.2 Metrics

Each experiment was repeated 3 times after a warm-up run, and we report the average of the 3 runs. We report the standard deviation (stdev) where applicable.

6.4.2.1 Interactivity Constraint Criteria

For each experiment, we tested whether the results were returned within an interactive response time limit, set to 500 ms [95]. We set the minimum number of results to be returned to be 500 in order to consider a system as being interactive to ensure that a reasonable number of tuples are available for the user to skim – unlike most related work, which set it at a single tuple. Although the interactivity criteria require us to deliver a limited amount of results within a limited time, we do not stop query execution after these were produced unless the client requests so.

6.4.2.2 Time for Completion

For each experiment, we report the execution time for all queries in seconds, which represents the time between the client issuing its first query and receiving all the results for all issued queries. This time is unrelated to the interactivity criteria, which is a binary criterion.

6.4.3 Experiments

6.4.3.1 Scale Of Simultaneous Queries

Traditional systems are limited in the amount of parallel computation they can provide since different clients use separate threads or processes. These systems utilize the idle time spent waiting for resources needed by a user to process other requests. If idle time is unavailable, they get overloaded. In our experiments, PostgreSQL and System X could not
process more than 128 and 512 parallel requests respectively. In contrast, our system was able to reach the maximum available threads allowed by the operating system.

![Diagram](image1)

**Figure 6.4:** Time To Completion, Data\(_{100MB \times 10MB}\), Machine\(_{4GB}\).

![Diagram](image2)

**Figure 6.5:** Time To Completion, Data\(_{1GB \times 1GB}\), Machine\(_{4GB}\).

In Figures 6.4 and 6.5, we compared the query execution times for different relation sizes with increasing number of concurrent queries. We have not reported the performance
of NL or CNL here since they are much slower than CHJ. We have presented the execution times for FJ and FFJ only for Data\(_{100MB \times 10MB}\) – the experiments provided later compare the cyclic algorithms specifically.

We conclude:

- CHJ offers the best scalability.
- The performance of FJ is comparable to some of the hash join implementations.
- The performance of FFJ is closer to CHJ than to any of the other cyclic join algorithms.

### 6.4.3.2 Impact of Data Size

We report whether each algorithm can be considered as interactive in Tables 6.1, 6.2, and 6.3 based on the interactivity criteria (Section 6.4.2) for data sizes ranging from 10 MB to 100 GB.

<table>
<thead>
<tr>
<th>Threads</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1K</th>
<th>2K</th>
</tr>
</thead>
<tbody>
<tr>
<td>HJ</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>System X HJ</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>PostgreSQL HJ</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>CHJ</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>System X NL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>PostgreSQL NL</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>NL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CNL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FJ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FFJ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.1: Interactivity Over Data\(_{100MB \times 10MB}\), Machine\(_{4GB}\) for simultaneous queries (threads). NL, CNL, FJ, and FFJ are the only interactive algorithms in this case.
Table 6.2: Interactivity Over Data$_{1GB\times1GB}$, Machine$_{4GB}$ for simultaneous queries (threads). Here, NL, FJ, and FFJ are the only interactive algorithms.

<table>
<thead>
<tr>
<th>Threads</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1K</th>
<th>2K</th>
</tr>
</thead>
<tbody>
<tr>
<td>HJ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>System X HJ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CHJ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>System X NL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FJ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FFJ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.3: Interactivity Over Data$_{100GB\times100GB}$, Machine$_{244GB}$ for simultaneous queries (threads). Here, CNL, FJ, and FFJ are the only algorithms that scale for large relations, while being interactive.

<table>
<thead>
<tr>
<th>Threads</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1K</th>
<th>4K</th>
<th>8K</th>
<th>16K</th>
</tr>
</thead>
<tbody>
<tr>
<td>HJ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>System X HJ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CHJ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>System X NL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CNL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FJ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FFJ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

HJ does not provide interactive response times in most cases, while FJ, FFJ, and NL usually do (for our engine, NL, CNL, and FJ always provide interactive response times). Interestingly, when HJ failed, the reuse of cached hashmaps allowed CHJ to complete. All hash joins require additional memory during execution. This has a detrimental effect on the experiments performed using Data$_{100GB\times100GB}$ even on machines such as Machine$_{244GB}$ – the entire memory is consumed and the hash join execution fails. In comparison, CNL does not consume additional memory, while FJ requires a limited additional amount of memory. As seen in our experiments, FJ and FFJ always provide interactive response time.
Figure 6.6: Join Types Comparison, Data_{10MB \times 10MB}, Machine_{4GB}. Here, FJ and FFJ represent a good tradeoff between CHJ (non-interactive) and CNL (interactive).

We conclude:

- The cyclic versions are preferable over the non-cyclic ones for both performance as well as interactivity.
- FJ, FFJ, and CNL are always interactive.

### 6.4.3.3 Impact of Fast FluxJoin and FluxJoin

Here, we focus on the performance of FJ and FFJ. Figure 6.6 shows that FJ is exponentially faster than CNL, yet slower than CHJ - making it a good intermediate point between the two when interactivity is needed. FFJ performance is similar to that of CHJ. Due to its longer execution times, we ran CNL for select few configurations. In Figure 6.7, we show that FFJ performs nearly as well as CHJ. Yet, unlike CHJ, FFJ provides interactive response time.

We conclude:

- FJ and FFJ are preferable over CNL.
Figure 6.7: Join Types Comparison, Data$_{1024MB \times 100MB}$ on Machine$_{244GB}$. Here, FFJ is ideal for interactivity.

- When memory is not limited, FFJ is the ideal algorithm - based on response time and performance.

- CHJ always has the least execution time.

6.5 Related Work

Interactive queries have been explained in [29] and in [132, 133], the authors delve into details regarding the need for this type of queries in depth. An interactive engine challenges the current state of the art systems with additional requirements such as limited response time and in-flux queries [47, 68, 144]. The current state of the art developments can be divided into two research areas – parallel scans and shared scans.

Parallel scans [52, 56, 143, 147] aim to improve the resource utilization. The main idea is to reduce the system response time by using available resources that are not utilized. Although we have not explored parallelization, a parallel cyclic scan, or a parallel join on latched blocks, it is an interesting avenue for future work.
Shared Servers [88] allow different queries to execute within the same host server process. It also enables transactions to be executed on different server processes while preserving ACID requirements. It is implemented using queues that register queries for execution. Each query sequentially uses the Shared Server process. In Shared Servers, one query executes at a time within each Shared Server process. When the current query completes, the next query is dequeued and begins execution. In contrast with Shared Servers, FluxQuery executes all the queries simultaneously, sharing the table scans, and not only the hosting process. The similarity between our approach and Shared Servers is mostly related to the queue for communication between the client and the execution process.

Sharing computation had first been proposed in the context of multi-query optimization (MQO) [130]. It aims to find common subexpressions between queries so as to execute them only once. It is especially suited for complex OLAP queries which contain multiple subexpressions. This approach has drawbacks of the queries needing to have been submitted close to each other, as well as subexpression detection being expensive.

Shared scan, introduced explicitly in [55] and based on [45, 58, 88, 146], introduces the ability to share a table scan for processing multiple queries. We adopted the shared scan approach since it is well suited for providing interactivity. The shared scan, as presented before, aims to provide system predictability in OLTP environments, whereas we focus on data warehouse environments. It originally targets a response time of 2s, which is greater than interactive thresholds of at most 0.5s. We add to the existing work a thorough discussion regarding joins including Cyclic Hash Join and Cyclic Nested Loop Join, which while supported earlier had not been expanded on. We demonstrate how sharing latches in addition to scans can further speed-up query execution. In addition, we utilize the client for
some of the server process operations (for example, building hashmaps). Furthermore, we start executing queries at the granularity of data blocks as opposed to the entire relation.


Multi-column join is an interesting avenue to pursue. The techniques shown in [110] allow implementing one join process that enables two or more column join. The same approach, as presented there can be extended to more than three columns, therefore we believe the expansion of our CHJ/CNL algorithms to join more than two columns is feasible and can be targeted in future work.

We implemented 4 different cyclic join algorithms: CNL, CHJ, FJ, and FFJ. Other join types such as sort-merge joins, etc. and aggregation types exist, which might also benefit from our approach. Some, for example, ripple joins [59], cannot be implemented in a cyclic manner since data is not scanned linearly in both tables.

The database engines, objects, and operators change over time as shown in [90, 110]. Our objects are built in a manner that allows tunneling an output resultset stream into another operator (pipelining), allowing our system to use techniques shown in [148]. Cyclic scans need to be researched in more depth with regards to pipelining. We find query vectorization and pipelining techniques, presented in MonetDB [21], useful towards that goal.

In Eddies [13], moments of symmetry is introduced which has resemblance with our notion of Safe Points. Both are similar in the action they perform - enforcing a halt of the main algorithm for communication (in Safe Points, query list modifications, in “Eddies” for join data intended for processing). However, the intention and implementation are vastly
different. Safe Points are meant for internal lists modifications and are used for assuring consistency of memory structures, allowing multiple simultaneous queries execution while lowering performance hit. Moments of Symmetry are meant for synchronization of data between nodes processing a singular distributed join assuring result correctness.

Shared Operators have been introduced before in the context of continuous queries over streams [98]. Assuming the subsetting criteria is identical, Shared operators allow executing specific filters simultaneously for multiple queries. In our work, not only are the join criteria different, but the columns the joins are on differ as well - preventing us from using Shared Operators. Our system is structured for using different subsettings and join criteria while sharing the table scans.

6.6 Conclusion & Future Work

In this work, we formalized the concept of an in-flux interactive query. We introduced new techniques for joining tables while also sharing the data and the latches. We developed protocols for adding queries to the shared join process, while other queries are executed, at a low cost. We proposed a hybrid join strategy called the FluxJoin, which reduces the complexity of the Nested Loop Join, while using limited memory and maintaining the short response time necessary for interactive queries. Fast FluxJoin, a variant of FluxJoin, which results in lower execution time at the cost of additional memory, has been presented as well. Our cyclic scan-based execution of likely queries bears a strong resemblance with the speculation used by DICE and Sesame. While all the three systems speculatively execute likely future queries, FluxQuery uses the results to show the user preview of possible query while DICE and Sesame use it to speed up the next user query.
The query intent model has not presented in detail since our focus is the cyclic scan-based execution. The model deserves work with the focus solely on it. A major performance bottleneck was the transfer of the result set to the client. Work related to processing data streams \([40, 137]\) can be used for handling this issue. It can be mitigated using sampling as well \([37, 114]\). Fault tolerance and scale out streams \([24]\) are also relevant to our work. There is a clear need to research optimization of cyclic engines. We have assumed that an optimizer, which builds execution plans for such engines exists, and we would like to work on it in the future – we currently manually built the execution plans. There are multiple aspects of sampling that can be used in the context of interactive query execution. The previous chapter shows that a row sampled while keeping a particular join combination in consideration is usually not useful for another join combination and, thus, sampling rows with multiple join criteria in mind is an interesting avenue for future work. Finding samples that would be relevant for all possible join and subsetting criteria is both challenging and interesting.
Chapter 7: Conclusion

7.1 Contributions

In this work, we have seen different sampling-based techniques that help query large datasets within interactive response times. DICE used speculative querying, result caching, distributed execution, and sampling to query a billion tuples within interactive response times. Its user interface has been built to accelerate its features using point and click approach. Sesame extends DICE by using an in-memory system to speculate not only on measures themselves but also on error estimates resulting in speedups of up to $25\times$. We have also proposed a unified correlation-based approach to handle the difficult problem of sampling joins. FluxQuery uses cyclic-scan based approach to process in-flux queries.

7.2 Limitations of Sampling

While sampling provides benefits such as reducing the computational resources and execution time, it does have its issues. Sampling might not be applicable to accuracy-critical fields such as medicine and finance – here, error in analyses might have severe consequences. Waiting longer for accurate results and using more computational resources might not be the critical concerns in such domains.
Further, sampling also brings sampling error into the process. This requires the analyst to consider the error as part of their decision making. It might also might lead them towards an incorrect decision path. Thus, while the analyst’s journey will end sooner, he is reaching the wrong destination.

Another drawback of sampling is that sampling semantics are unclear for complex SQL queries. In particular, the meaning of error is unclear – should missing groups in the result and the statistical error of the measure be combined to present a unified error to the user? Results of a sub-query being input into a larger query increases the difficulty of this problem.

7.3 Applicability of Sampling

While sampling might not be applicable in accuracy critical fields, it can be applied to other areas, where result error might be tolerable. We look at some of these use cases.

One of the factors driving sampling recently has been increasing usage of user interfaces in the querying process. The result error gets translated into pixels, where it might be insignificant, due to the limited screen resolution.

Machine learning-based tasks such as providing search results are suitable for sampling as well, where low latency is critical in user engagement. In such cases, ground truth does not exist, with results being based on a complex ranking function.

Exploratory data analysis and sense-making involve a user trying to navigate and understand the data. He is not looking to perform a deep dive of the data, but only to familiarize with it. In such cases, giving results quickly is of more utility to the user than a slow query that gives perfect results.

Hypothesis testing involving ruling things out (or in) is also suited to sampling. If the probability that a condition might be ruled out for the entire dataset is low, it is likely that it
will not be much higher for a smaller dataset. If the operator that generates the dataset is blocking, it might result in a significant waste of computational resources. If the hypothesis does not get ruled out, the sample size can be increased.

7.4 Test of Time

Amongst the different ideas proposed in this work, we believe our proposal for interactive exploration using *speculation* will stand the test of time. Materializing an entire cube offline is an expensive operation. Speculation allows us to perform partial cube materialization in an online fashion, thereby removing the need for the high offline cost, while reducing the query latency. One academic metric for the strength of a concept is its influence on the community, which has been demonstrated by multiple projects based on the notion of speculation being spawned by other research groups. We are currently building the speculation idea into a middle layer, which we hope will have an industrial impact as well.

7.5 Future Work

Interactive visualizations can accelerate the data analysis loop through near-instantaneous feedback. To achieve interactivity, techniques such as sampling and data cubes are typically employed. While data cubes can speedup querying for moderate-sized datasets, they are ineffective at doing so at a larger scales due to the size of the materialized data cubes. On the other hand, while sampling can help scale to large datasets, it adds sampling error into the process. While increasing accuracy by looking at more data may sometimes be valuable, providing result minutiae might not be necessary if they do not impart additional significant information. Indeed, such details not only incur a higher *computational* cost, but also tax the *cognitive* load of the analyst with worthless trivia. It is indeed possible to use data binning
and provide error-free results for large datasets by increasing bin resolutions progressively over time. This looks like an interesting avenue for future work.

User guidance systems try to understand user behavior through cues such as mouse movements, clicks, and queries. These cues constitute user’s *Query Intent* and can be used to improve prediction for the next user query. Our previous experience [77, 81] in designing interactive visualizations has shown us that users spend significantly more time looking at the results trying to gain insights, than querying. Consequently, the system is ignorant of a large fraction of possibly available user signal. In the future, we plan to utilize a hitherto unexplored user input mode in interactive visualizations to rectify this issue – *User Gaze*. While mouse or keyboard-based user interaction represents the query intent, user gaze represents the *Result Intent*. Understanding result intent improves user guidance as it shows which visualizations the user is interested in.
Bibliography


[92] Xiaolei Li, Jiawei Han, Zhijun Yin, Jae-Gil Lee, and Yizhou Sun. Sampling cube: A framework for statistical olap over sampling data. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD ’08, pages 779–790, New York, NY, USA, 2008. ACM.


[141] Fangbo Tao, Kin Hou Lei, Jiawei Han, Chengxiang Zhai, Xiao Cheng, Marina Danilevsky, Nihit Desai, Bolin Ding, Jing Ge Ge, Heng Ji, Rucha Kanade, Anne Kao, Qi Li, Yanen Li, Cindy Lin, Jialu Liu, Nikunj Oza, Ashok Srivastava, Rod Tjoelker, Chi Wang, Duo Zhang, and Bo Zhao. Eventcube: Multi-dimensional search and mining of structured and text data. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’13, pages 1494–1497, New York, NY, USA, 2013. ACM.


