EFFICIENT QUERY PROCESSING IN MCDB-R

By

RAVINDRANATH CHOWDARY JAMPANI

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY UNIVERSITY OF FLORIDA 2012
Dedicated to my family, friends and teachers
ACKNOWLEDGMENTS

I would like to thank my advisors, Chris and Alin. I have been fortunate to have them as my advisors, and this work would not be possible without their guidance and support. Their vast knowledge, quick thinking, great insight into the problems and their enthusiasm for research and teaching have been inspiring. It has been great working with them. I am thankful to Chris for letting me work on my thesis project. I am grateful to the Department of Computer & Information Sciences & Engineering (CISE), University of Florida and my advisors for supporting me throughout my studies.

I am thankful to Dr. Ungor for letting me work on the meshing project. This project had been a great learning experience. I am thankful to my colleagues and friends at CISE – Mingxi, Subi, Nirmalya, Manu, Niketan, Lixia, Manas, Luis, Fei, Ferhat, Florin, Manna, Vishak, and Amit. I am grateful to Subi Arumugam and Risi Thonangi for their conversations related to research and implementation of database systems. Special thanks to the students who attended my Microsoft Office course in the Fall of 2010. They were a great motivation for me to put in the necessary effort to prepare for and teach the topics. Their feedback during the course was also very useful. It remains a unique teaching experience for me.

I would also like to mention the International Institute of Information Technology (IIIT), Hyderabad, India, for providing me with a great learning environment during my undergraduate studies. Thanks to Dr. Govindarajulu Regeti, and Dr. Rajeev Sangal for inspiring me to pursue a doctorate and to Dr. Kamal Karlapalem, Dr. Sushma Bendre and Dr. Prosenjit Gupta for helping me in my application process.

I am grateful to my friends from my undergraduate days: Sashi, Srikanth, Rishi, Santhosh, Jhansi, and Teja. They have been a source of constant support. I am fortunate to have formed friendships with Tapasvi, Manu, Niketan, Subhajit, Arun, Ganesh, Mriganka, Siva, Aswin, Jayendra and Muthu during my graduate studies.
Gainesville was a fun place to stay because of them. Finally, I would like to thank my family, whose continued encouragement has made this effort possible.
TABLE OF CONTENTS

ACKNOWLEDGMENTS ......................................................... 4
LIST OF TABLES .......................................................... 8
LIST OF FIGURES .......................................................... 9
ABSTRACT ................................................................. 11

CHAPTER

1 INTRODUCTION ......................................................... 13
  1.1 Main Contributions ................................................ 17
     1.1.1 Problem 1: Serializing Variable Generating Functions ...... 17
     1.1.2 Problem 2: Finding an Extreme Sample ....................... 18
     1.1.3 Problem 3: Incorporating Anti-Join .......................... 19
  1.2 Organization ....................................................... 20

2 MCDB-R OVERVIEW .................................................. 21
  2.1 Background ......................................................... 27
     2.1.1 Rare Event Simulation ....................................... 27
     2.1.2 Gibbs Sampling .............................................. 28
  2.2 Schema Specification ............................................... 29
  2.3 The Variable Generating Function ................................. 32
  2.4 The Seed Operator ................................................ 34
  2.5 The Instantiate operator .......................................... 35
  2.6 Gibbs Tuple Bundles ............................................... 37
     2.6.1 Tail-Sampling-Seed and Seed Queue ......................... 39
     2.6.2 Split Operator .............................................. 40
  2.7 Gibbs Looper and Tail Sampler ................................... 42
     2.7.1 Gibbs Queue .............................................. 42
     2.7.2 Rerunning the Query Plan .................................. 44
     2.7.3 Materializing the Operators ................................. 45

3 RELATED WORK ....................................................... 48
  3.1 Statistical Operations in Relational Databases ................. 48
  3.2 Probabilistic Databases .......................................... 48
  3.3 Top-k Ranking in Probabilistic Databases ....................... 51
  3.4 Monte Carlo Methods in Databases, Data Mining, and Machine Learning 53
  3.5 Risk Analysis Software .......................................... 56
4 SERIALIZING SAMPLE GENERATION PROCESS ........................................... 57
  4.1 Background .................................................................................. 57
  4.2 Motivation .................................................................................... 58
  4.3 Solution Outline .......................................................................... 61
  4.4 Serialization .................................................................................. 61
    4.4.1 Incorporating Serialization in a Query Plan ................................. 64
    4.4.2 Stochastic Join Predicates ....................................................... 64
  4.5 Efficient De-serialization ................................................................. 66
  4.6 Related Work .............................................................................. 67
  4.7 Experiments .................................................................................. 68
5 REPLACING AN EXTREME SAMPLE ......................................................... 74
  5.1 Motivation .................................................................................... 74
  5.2 Solution Outline .......................................................................... 76
  5.3 Filtering ........................................................................................ 77
  5.4 Fragmenting a Large Tuple Bundle ............................................... 79
  5.5 Implementation Details ................................................................. 80
    5.5.1 TSSeed and Sample Generation ............................................. 82
    5.5.2 How to Identify an Extreme Sample? ..................................... 83
  5.6 Experiments .................................................................................. 85
6 ANTI JOIN .................................................................................................. 92
  6.1 Motivation .................................................................................... 92
  6.2 Composite Attribute Formats ....................................................... 95
    6.2.1 Long Format ........................................................................ 95
    6.2.2 Short Format ....................................................................... 98
  6.3 Discussion on Extreme Samples ................................................... 100
  6.4 For Queries with Multiple Anti-Joins ........................................... 101
  6.5 Experiments .................................................................................. 101
7 CONCLUSION ............................................................................................ 106
APPENDIX: QUERIES ................................................................................... 107
REFERENCES .............................................................................................. 111
BIOGRAPHICAL SKETCH ............................................................................. 116
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Example: Flaw of averages</td>
<td>14</td>
</tr>
<tr>
<td>4-1</td>
<td>Wall clock execution times</td>
<td>71</td>
</tr>
<tr>
<td>4-2</td>
<td>Disk I/O and sample statistics</td>
<td>71</td>
</tr>
<tr>
<td>4-3</td>
<td>Scaling with database size</td>
<td>71</td>
</tr>
<tr>
<td>4-4</td>
<td>Scaling with DB instances (time in seconds)</td>
<td>72</td>
</tr>
<tr>
<td>4-5</td>
<td>Parallel sample generation</td>
<td>72</td>
</tr>
<tr>
<td>4-6</td>
<td>Performance for different VG functions (time in minutes)</td>
<td>72</td>
</tr>
<tr>
<td>5-1</td>
<td>Poisson random relation</td>
<td>75</td>
</tr>
<tr>
<td>5-2</td>
<td>After a phase of rejection sampler</td>
<td>75</td>
</tr>
<tr>
<td>5-3</td>
<td>Wall clock running times (in seconds)</td>
<td>89</td>
</tr>
<tr>
<td>5-4</td>
<td>Number of extreme samples identified</td>
<td>89</td>
</tr>
<tr>
<td>5-5</td>
<td>Performance with increasing input size (in minutes)</td>
<td>89</td>
</tr>
<tr>
<td>5-6</td>
<td>Samples generated (in Billions)</td>
<td>89</td>
</tr>
<tr>
<td>5-7</td>
<td>Query reruns and extreme samples</td>
<td>89</td>
</tr>
<tr>
<td>5-8</td>
<td>Performance with change in aggregate attribute</td>
<td>89</td>
</tr>
<tr>
<td>5-9</td>
<td>Performance with changing max. samples (in minutes)</td>
<td>89</td>
</tr>
<tr>
<td>5-10</td>
<td>Execution parameters with changing max. samples</td>
<td>89</td>
</tr>
<tr>
<td>5-11</td>
<td>Number of extreme samples with increasing Gibbs iterations</td>
<td>90</td>
</tr>
<tr>
<td>6-1</td>
<td>Performance with varying seeds per tuple (in minutes)</td>
<td>103</td>
</tr>
<tr>
<td>6-2</td>
<td>Performance on large datasets (in minutes)</td>
<td>104</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>CustOrder table: (a) Deterministic, and (b) Uncertain</td>
<td>22</td>
</tr>
<tr>
<td>2-2</td>
<td>Stochastic CustOrder table</td>
<td>22</td>
</tr>
<tr>
<td>2-3</td>
<td>Multiple instances of CustOrder after Monte Carlo sampling</td>
<td>24</td>
</tr>
<tr>
<td>2-4</td>
<td>Query-result histogram</td>
<td>24</td>
</tr>
<tr>
<td>2-5</td>
<td>Gibbs cloning: Generating samples</td>
<td>25</td>
</tr>
<tr>
<td>2-6</td>
<td>Gibbs cloning: Deleting lower 50%</td>
<td>25</td>
</tr>
<tr>
<td>2-7</td>
<td>Gibbs cloning: Clone the elite samples</td>
<td>26</td>
</tr>
<tr>
<td>2-8</td>
<td>Gibbs cloning: After perturbation with Gibbs sampler</td>
<td>27</td>
</tr>
<tr>
<td>2-9</td>
<td>An example Seed attribute</td>
<td>34</td>
</tr>
<tr>
<td>2-10</td>
<td>Instantiate Operator</td>
<td>36</td>
</tr>
<tr>
<td>2-11</td>
<td>Tail sampler operator</td>
<td>38</td>
</tr>
<tr>
<td>2-12</td>
<td>CustOrder with Gibbs tuples</td>
<td>39</td>
</tr>
<tr>
<td>2-13</td>
<td>Example: TSSeed and Seed</td>
<td>40</td>
</tr>
<tr>
<td>2-14</td>
<td>Stochastic table in a join</td>
<td>40</td>
</tr>
<tr>
<td>2-15</td>
<td>Stochastic table after a Split</td>
<td>41</td>
</tr>
<tr>
<td>2-16</td>
<td>Join result tuples</td>
<td>41</td>
</tr>
<tr>
<td>2-17</td>
<td>Gibbs queue</td>
<td>43</td>
</tr>
<tr>
<td>2-18</td>
<td>Example: End of run 1</td>
<td>44</td>
</tr>
<tr>
<td>2-19</td>
<td>Example: Beginning of run 2</td>
<td>45</td>
</tr>
<tr>
<td>2-20</td>
<td>Query plan before materialization</td>
<td>46</td>
</tr>
<tr>
<td>2-21</td>
<td>Query plan after materialization</td>
<td>46</td>
</tr>
<tr>
<td>2-22</td>
<td>Query plan with multiple reruns</td>
<td>47</td>
</tr>
<tr>
<td>4-1</td>
<td>An MCDB-R query plan</td>
<td>57</td>
</tr>
<tr>
<td>4-2</td>
<td>Modified query plan</td>
<td>60</td>
</tr>
<tr>
<td>4-3</td>
<td>A VG function process</td>
<td>62</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>4-4</td>
<td>Serialization of VG function</td>
<td>62</td>
</tr>
<tr>
<td>4-5</td>
<td>Bayesian VG: Input serialization</td>
<td>63</td>
</tr>
<tr>
<td>4-6</td>
<td>Bayesian VG: Inner input</td>
<td>63</td>
</tr>
<tr>
<td>4-7</td>
<td>Bayesian VG: Outer input</td>
<td>63</td>
</tr>
<tr>
<td>4-8</td>
<td>Serialization of Bayesian VG input process</td>
<td>64</td>
</tr>
<tr>
<td>4-9</td>
<td>Join on stochastic attribute on one side</td>
<td>65</td>
</tr>
<tr>
<td>4-10</td>
<td>Query plan after serialization</td>
<td>65</td>
</tr>
<tr>
<td>4-11</td>
<td>Join on stochastic attributes on both sides</td>
<td>66</td>
</tr>
<tr>
<td>4-12</td>
<td>Query plan after serialization</td>
<td>67</td>
</tr>
<tr>
<td>5-1</td>
<td>Query plan</td>
<td>78</td>
</tr>
<tr>
<td>5-2</td>
<td>Modified plan with filtering</td>
<td>78</td>
</tr>
<tr>
<td>5-3</td>
<td>Splitting a large tuple bundle</td>
<td>79</td>
</tr>
<tr>
<td>5-4</td>
<td>Query plan with Sample router: normal mode</td>
<td>80</td>
</tr>
<tr>
<td>5-5</td>
<td>Query plan with Sample router: active mode</td>
<td>81</td>
</tr>
<tr>
<td>5-6</td>
<td>A TSSeed and assigned sample values</td>
<td>82</td>
</tr>
<tr>
<td>5-7</td>
<td>TSSeed and assigned samples after a Gibbs sample iteration</td>
<td>82</td>
</tr>
<tr>
<td>6-1</td>
<td>Example: Anti-Join in a relational database</td>
<td>92</td>
</tr>
<tr>
<td>6-2</td>
<td>Example: Stochastic table</td>
<td>94</td>
</tr>
<tr>
<td>6-3</td>
<td>Example: After Split</td>
<td>95</td>
</tr>
<tr>
<td>6-4</td>
<td>Composite Attribute: Long format</td>
<td>96</td>
</tr>
<tr>
<td>6-5</td>
<td>Long Format: Tuples after Anti-join</td>
<td>97</td>
</tr>
<tr>
<td>6-6</td>
<td>Composite Attribute: Short format</td>
<td>98</td>
</tr>
<tr>
<td>6-7</td>
<td>Short Format: Tuples after Anti-join</td>
<td>99</td>
</tr>
</tbody>
</table>
Quantitative risk analysis has become an integral part of large financial institutions. For banking and insurance enterprises, calculating the risk of future insolvency is mandatory per the regulatory directives such as Solvency and Basel. Performing risk analysis in these enterprises is difficult because the systems are highly complex and have a large number of uncertain variables. One form of risk analysis is in understanding the impact of extreme events that have a low chance of occurrence. The common methods used to estimate this extreme risk come from statistics and rare event simulation fields, and are based on Monte Carlo sampling. However, these sampling-based methods produce huge amounts of data. Therefore, the current implementations of risk analysis methods have scalability problems, particularly as they relate to complex systems with a very large number of variables.

Monte Carlo Database with Risk analysis (MCDB-R) is a database system designed to facilitate scalable risk analysis on large datasets. MCDB-R combines sampling-based statistical methods and relational database technology for scalability on large uncertain data sets. Uncertainties in data are modeled as probabilistic distributions. A query on this uncertain data with probability distributions results in another probabilistic distribution called query-result distribution. For a given scenario defined by a query, analyzing its extreme events requires examining the upper or lower areas of the query-result distribution, where the events occur with very low probability.
The specific methods used in MCDB-R for this analysis are: Gibbs sampling from statistics and cloning from rare-event simulation. These methods are used to efficiently sample from the low probability areas of the query-result distribution for the analysis.

This dissertation discusses three improvements to the query processor in MCDB-R. The first technique provides a mechanism to move the sample generation (during the query execution) closer to the location where those samples are actually processed. Huge number of samples are generated during the query execution in MCDB-R, and this new mechanism reduces the sample movement in the memory hierarchy. Response time of the queries is improved significantly; sometimes by an order of magnitude, as shown in the experiments.

MCDB-R employs a rejection algorithm at the core of its sampling-based risk analysis. For each uncertain variable, the rejection algorithm looks at a series of samples and discards them unless the sample fits a given constraint. Sometimes the constraints are so stringent that the rejection algorithm needs to process millions of samples before finding a good fit. In MCDB-R, an instance like this will require the same number of samples to be produced for all variables. In the second technique, this problematic instance is isolated from the rest of the query execution and then run separately to find an acceptable fit. The normal execution restarts after finding the required sample.

Finally, adding an anti-join operator to the MCDB-R execution engine is explained. This new operator enables the system to execute subset-based queries with not-in and not-exists clauses. Performing an anti-join in this system is not trivial because of the stochastic nature of the data. The system does not know which samples are actually used until the end of the query execution. Two methods to implement the anti-join operator are discussed and then compared through experiments.
CHAPTER 1
INTRODUCTION

The 2008 financial crisis is considered to be the worst such crisis since the Great Depression [70]. It resulted in the collapse of banks and insurance corporations, which necessitated government bailouts and resulted in plunging stock markets. The trigger for this financial apocalypse was the subprime mortgage crisis, which caught financial institutions off guard until it was too late. Institutions learned that underestimating the risk of extreme events and the liquidity required to counter those events left them in a very precarious situation. The crisis demonstrated that low probability and high-impact events do happen. The current risk management models do not place sufficient emphasis on extreme events, which is a critical issue [35]. The impact of extreme risk events can be lowered if they are detected early and managed properly. However, due to the lack of preparation, the impact can be greater than necessary.

Another issue with current risk management models is their excessive approximation. Current financial system is highly dynamic and complex due to globalization. Parts of this complex system are either approximated or, are totally ignored in order to reduce the number of variables in the model. The approximations are obtained by using averages over multiple variables. One of the main reasons for reducing the number of variables is due to the limitations of the software that is currently used, which is based on spreadsheets, and is not scalable for more fine-grained models.

The problem that arises due to imprudent approximation with averages can be explained by using the simple example from [55], Chapter 19. In this example, a bank is estimating the average profit it might derive from its loans. Assume that the bank pays 4% interest per year on the money it uses. It lends to the customers with good credit at a 6% rate, therefore, the profit margin is 2%. For customers who are less credit-worthy, the bank loans money at 14%, yielding a 10% profit margin. The cost overhead to manage the loan is $25 per year. Thus, an account of $600, with a margin of 6% will
<table>
<thead>
<tr>
<th>Account with Good Credit</th>
<th>Balance</th>
<th>Margin</th>
<th>Net Revenue</th>
<th>Profit after Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1000</td>
<td>2%</td>
<td>20</td>
<td>-5</td>
</tr>
<tr>
<td>Account with Bad Credit</td>
<td>200</td>
<td>10%</td>
<td>20</td>
<td>-5</td>
</tr>
<tr>
<td>Correct Average</td>
<td>600</td>
<td>6%</td>
<td>20</td>
<td>-5</td>
</tr>
<tr>
<td>Flawed Average</td>
<td>600</td>
<td>6%</td>
<td>36</td>
<td>11</td>
</tr>
</tbody>
</table>

yield a profit of $11 ((6% of $600) - $25 = $11). Refer to the data given in Table 1-1. If we consider both groups of customers separately, we will obtain an accurate estimate. A good customer with a $1000 balance on a 2% margin will generate $20 revenue and a -$5 in total profit. The other customer with a balance of $200, and a margin of 10% will again generate $20 in revenue and a -$5 in total profit. The correct estimate for the average profit is provided in Table 1-1, row 3. However, if we try to estimate the average profit based on the average balance of $600 (($1000+$200)/2) and an average margin of 6% ((2%+10%)/2), the average net revenue is $36 and the average profit is $11, as shown in Table 1-1, row 4.

Simulation based on sampling is an excellent way to avoid the problem created by using averages. However, sampling is very data intensive and scalability becomes an issue as a result of the increase in variables. The MCDB-R system is built to provide scalable Monte Carlo simulations for extreme risk analysis. Scalability is achieved through the use of existing database technology, and incorporating the Monte Carlo simulations internally in the query processing.

The advantages of using MCDB-R over existing spreadsheet based modeling includes:
Scalable simulations – The scalability of MCDB-R facilitates Monte Carlo simulations over a finer and richer model, avoiding the issues with approximation using averages. A fine-grained model is important to obtain more accurate results and avoiding the flaws introduced by averages.

Ease of use – Users of spreadsheet applications such as Microsoft Excel must first extract the data from a database through aggregate queries (such as AVG, COUNT, and so forth) and then load the data into the application. These two steps require that the user buy and learn several software packages, involving both considerable money and time. MCDB-R avoids these issues by providing Monte Carlo simulations within the database.

Next, we give an example 'what-if' query, and how MCDB-R analyzes extreme risk on that query.

Example: Let us examine the following what-if query from [37]: "What would the profits be if we raised all prices by 5% this year?". For this query, relevant information like fluctuation in demand with a price increase is unknown. Therefore, [37] uses a Bayesian model to calculate the fluctuation in demand as a parametric probability density function. The attribute for the new demand is called a stochastic attribute as its value is defined by a probability distribution. The whole database itself is now stochastic, and is characterized by a very high dimensional joint distribution. A query on this stochastic database will result in another distribution instead of a single value as in a deterministic database. This result distribution is called a query-result distribution. Obtaining this query result distribution is not simple because the data is high-dimensional, making a direct analytical solution very difficult. For this reason, the answer is calculated through Monte Carlo-based sampling [51] in the following way: we generate multiple new demand values from the probability-density functions. Each set of new demand values will create an instance of the database. The query is executed on each instance of the database. The output of the query is a set of profit values from the query-result distribution. This is the method used in the Monte Carlo Database System (MCDB) [37]. For efficient query processing, MCDB bundles all values of a
single stochastic tuple (from different database instances) together into a single tuple bundle.

Let us analyze the extreme risk for the 5% price increase 'what-if' query to understand the worst possible scenarios for such a decision. For example, examining the lowest possible total profit value that can occur would be of interest. In the above Bayesian model the worst case can be defined as the value of total profit in the $0.001$ quantile of the query-result distribution (this low probability region is generally called the tail of a distribution function). We can better understand the risk of the decision with the knowledge of such worst-case scenarios. A large number of samples in the $0.001$ quantile space will be useful in estimating the profit value accurately and also in determining what lead to such poor profits. A decision-maker could then decide on the strategies to take to counter such effects.

One approach for obtaining samples in the tail is to generate large numbers of samples on the entire event space using MCDB and then select only those that fall in the tail region. If the samples are not requested from the extremely low probability regions, then MCDB works well. Generating 100 samples from the $0.01$ quantile is not very costly because generating 10,000 samples from the whole distribution is not expensive. However, this approach is quite inefficient if we are looking more deeply in the tail, such as the $0.000001$ quantile. In many applications, an event with a probability of $10^{-6}$ or less is called rare. Therefore, getting 100 samples of interest in such applications requires $10^8$ samples in the full-event space, which is not practical in MCDB. MCDB-R is designed to perform this 'tail-sampling' task efficiently. MCDB-R, in addition to utilizing tuple bundle and variable-generating (VG) function framework from MCDB, uses Gibbs cloning ideas from rare-event simulation literature. The Gibbs cloning method is used to directly sample from the tail of the distribution and, therefore, decreases the overall cost of the query. Gibbs cloning is an iterative process to move further into the tail of the distribution function. At each step we delete the uninteresting
samples and replace them with exact copies of interesting samples (a cloning step). The new set of samples will be highly correlated due to the cloning. Hence, these samples are randomly perturbed so that the new samples after the perturbation are reasonably uncorrelated. This perturbation is performed by the Gibbs sampler because of the high dimensionality of the data.

1.1 Main Contributions

Our group built a prototype of the MCDB-R system, and we tested its performance on various queries (including the example what-if query described above). We found that the system was not as efficient as we expected. After profiling the system, we discovered two issues:

- Moving large sample arrays in and out of memory is taking most of the execution time.
- The system is generating a large number of samples that are not even used.

As part of this dissertation, I am providing solutions to mitigate these issues. Additionally, I am tackling a third problem: to implement an Anti-join operator in MCDB-R. The Anti-join operator is important because it enables the system to run sub-queries with \textbf{NOT IN} and \textbf{NOT EXISTS} clauses. My dissertation also discusses the evaluation and implementation of the solutions that I proposed for all three problems, which are briefly described in the following subsections.

1.1.1 Problem 1: Serializing Variable Generating Functions

The sample sizes in MCDB-R are very large because of the selective nature of the rejection algorithm during the Gibbs sampling process (explained in Chapter 2). Using tuple bundles to group the samples together and store as an array results in faster processing. However, tuple bundles are extremely large, and moving them in the memory hierarchy is very time consuming. I present a technique that enables the system to serialize the sample generation process and to delay the actual sample generation as long as possible. This technique avoids the movement of a large numbers
of samples (or tuple bundles) through some expensive operations such as sorts and joins. For example, in the current system, we executed one of the benchmark queries (Q2 from Section 4.7) on a 200 megabytes (MB) dataset with 50,000 tuples with uncertain (or stochastic) data and generated result samples in the 0.999 quantile. The benchmark query writes and reads more than a terabyte (TB) to and from the disk during its execution, and generates more than 3 billion samples on average before it completes. The large sample sizes are due to some stochastic tuples that require a few samples with probability in the order of $10^{-3}$ or less. For example, it is possible for the query to look for a sample in the order of $10^{-5}$ probability. In the existing system, this task requires generating $10^5$ samples on average for each required tuple, and passing the samples through the query plan.

In simple queries with no complex joins on stochastic attributes (like the what-if query mentioned above), the sample generation can be delayed until the top-most operator in the query plan. In the current system, when a query is out of samples, the query is rerun with a larger set of samples. For some queries, the method that I propose (described in Section 4.4) also eliminates the requirement of rerunning the query plan when more samples are needed. Even in queries with joins on stochastic attributes, my method can eliminate the execution of some of the expensive operators before that join, which speeds up the query execution from the second rerun onwards.

1.1.2 Problem 2: Finding an Extreme Sample

Gibbs sampler processes an enormous number of samples during a single query execution instance. Therefore, the chances of the occurrence of an extreme sample (one that occurs with very low probability) in the sample stream is good. Such a sample is easily accepted if it is at the correct side of the tail. However, replacing that sample during the next Gibbs sampler phase may require another sample of nearly equal value. Replacing one such extreme sample could result in numerous query reruns and
increase the system response time. In Chapter 5, I look at an effective solution for this problem.

My solution has two parts. First, the query plan is rerun with only relevant tuples and then it filters out everything else. This method reduces the data flow in the system. Second, for that specific tuple, a large number of samples are generated. The larger number of samples will increase the chances of finding a replacement sample, and, hopefully, eliminate additional reruns for this tuple. One specific extreme sample might need tens of millions of samples to find a replacement. Since all the samples cannot fit on a single page, I developed a successful workaround. The large sample array in a tuple is broken into numerous smaller arrays as they are being generated. These smaller tuples (each storing one small sample array) are passed through the query plan until they reach the Gibbs sampler where they can be recombined together. The sequence identifiers are stored to remember the order in which the smaller tuples need to be grouped together when the tuples are being used.

1.1.3 Problem 3: Incorporating Anti-Join

Anti-join of relations R and S (R \( \bowtie \) S) returns tuples in R that do not match tuples in S on the common attributes. In a relational database, this operation is straightforward as its execution is similar to an equality-join. However, in MCDB-R it is difficult due to the delay of the actual execution of the predicate until the end of the query plan. A stochastic attribute participating in the Anti-join from the right relation S introduces complications: advancing the sample in S could result in the tuple in R being dropped. Consequently, the Gibbs sampling (refer to Chapter 2) must have all of the tuples from S that match the tuple in R. A simple but effective solution is to store all of the matching tuples from S in the corresponding tuple in R. During the Anti-join operation all the information necessary from matching tuples in S is added to the tuple in R. I also describe another solution that does not require storing all of the information from
matching tuples. This method eliminates the drawback of the first solution: the existence of huge tuple sizes as a result of storing attributes from all matching tuples.

1.2 Organization

This dissertation is organized as follows: Chapter 2 provides an overview of the system, the mathematical background required for MCDB-R, the Gibbs Sampler, and cloning. Chapter 3 describes prior work related to MCDB-R. Chapter 4, explains the serialization and de-serialization of the sample-generation process. Chapter 5 discusses the extreme sample problem. Chapter 6 explains the two methods by which the Anti-join can be added to MCDB-R.
MCDB-R is a system that facilitates risk analysis on high-dimensional uncertain data [6]. Performing risk analysis requires answering queries about risky rare events i.e. those that occur with very low probability but could have huge impact if they do occur. Data uncertainty in MCDB-R is modeled as probability distributions. The behavior of uncertain variables is captured by probability density functions. Risk analysis on such data can be done by observing the low probability regions – the upper and lower tails of a query result distribution function. For a given query, analytical forms of either the result distribution or any estimates on the result distribution is very unreliable and difficult to solve on high-dimensional distributions. Therefore, MCDB-R uses Monte Carlo sampling to approximately calculate statistics on the result distribution. While using the Monte Carlo methods, the uncertain database in MCDB-R is represented by multiple possible database (DB) instances (or database samples). These possible DB instances are generally called "possible worlds" in probabilistic databases literature [18]. Each instance is created by replacing each uncertain variable by a sample from its density function. Samples of an uncertain variable in different DB instances are independent of each other, even though they are identically distributed. This is made possible by using pseudo-random number generators for sampling. Now, running an aggregate query over the uncertain database returns multiple output values, one for each DB instance. The number of DB instances used is given as a parameter while specifying the query. MCDB-R is developed for generating samples from the tail of the query-result distribution. It adapts a technique called Gibbs cloning from rare event simulation to perform this task efficiently. Gibbs cloning method is used to sample from the tail of the distribution, and hence decreases the overall cost of the query. It is an iterative process that moves further into the tail of the distribution function after each iteration. Each iteration has a cloning step and a perturbation step. In a cloning step we delete
the samples that are away from the tail and replace them with exact copies of samples that are further into the tail. The new set of samples will be highly correlated due to the cloning. Hence, these samples are randomly perturbed such that the new samples after perturbation are reasonably uncorrelated. Due to the high-dimensionality of the data, Gibbs sampler is used for perturbation.

This chapter gives an overview of query processing in MCDB-R. First, we provide a high level understanding of risk analysis in MCDB-R using an example. We then provide relevant Monte Carlo and rare event simulation background required to understand the system. Next, we describe the query language in this system. Finally, different operators in MCDB-R are explained. We only explain operators that are fairly different from those in a standard relational database system.

<table>
<thead>
<tr>
<th>CUST ID</th>
<th>ITEM</th>
<th>QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>2</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>CUST ID</th>
<th>ITEM</th>
<th>NEW_QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>??</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>??</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>??</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>??</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>??</td>
</tr>
</tbody>
</table>

(b)

Figure 2-1. CustOrder table: (a) Deterministic, and (b) Uncertain

<table>
<thead>
<tr>
<th>CUST ID</th>
<th>ITEM</th>
<th>NEW_QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>DblGamma(α₁, β₁, α'₁, β'₁)</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>DblGamma(α₂, β₂, α'₂, β'₂)</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>DblGamma(α₃, β₃, α'₃, β'₃)</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>DblGamma(α₄, β₄, α'₄, β'₄)</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>DblGamma(α₅, β₅, α'₅, β'₅)</td>
</tr>
</tbody>
</table>

Figure 2-2. Stochastic CustOrder table
Example: Let us revisit the example query from Chapter 1: "what would the profits be if we raised all the prices by 5% this year?". Assume following tables in the database: \( \text{CustOrder}(\text{CUST ID, ITEM, QTY}) \), \( \text{Price}(\text{ITEM, PRICE}) \). In the table \( \text{CustOrder} \), \( \text{CUST ID} \) is the key, and \( \text{QTY} \) is the number of textttITEMs bought by the specific customer. The new demand from each customer for each item (call it \texttt{NEW QTY}) is not known after the price change. The deterministic table \( \text{CustOrder} \) and the new uncertain version of it with the attribute \texttt{NEW QTY} are shown in Figure 2-1. Since the values for \texttt{NEW QTY} are unknown, they are denoted by "??". A Bayesian model is used to estimate the new demand. The attribute \texttt{NEW QTY} is called stochastic attribute since it does not have a fixed value, and is defined by a probability density function. In this specific setting, we use two Gamma functions as the representative model for the attribute \texttt{NEW QTY}. Since the \texttt{NEW QTY} attribute takes integer values, the floating point sample value coming out of the Gamma functions will be rounded to the nearest integer. The new stochastic relation \( \text{CustOrder} \) is shown in Figure 2-2. Notice that the parameters for the DblGamma function can be different for each tuple.

The result of the query executed on a stochastic database will also be a probability distribution. Analytically solving for the result distribution is difficult due to high-dimensionality of the database. So MCDB [37] uses Monte Carlo sampling to answer the query. Multiple instances of the stochastic relation are generated using sampling. Sample instances are independent of each other. This step of sampling is depicted in Figure 2-3. In this figure the number of instances created is \( N \). The query when executed on these DB instances gives out one result for each DB instance. These obtained results can be considered as a independent and identically distributed samples from the query-result distribution. The final result consists of \( N \) values, and can be represented as a histogram as shown in Figure 2-4. If the query is run separately on each DB instance separately, then the system will be very slow. This performance hit is managed in MCDB by the
Figure 2-3. Multiple instances of CustOrder after Monte Carlo sampling

(grouping of all instances of a tuple together, called a tuple bundle, and running the query only once for all the DB instances.)

Figure 2-4. Query-result histogram
Now, let's consider performing extreme risk analysis on this decision query (increasing prices by 5%). For that, we want to know the worst possible consequences for such a price increase. For example, the lowest or highest possible total profit value that can occur with a probability of 0.001 is of interest. We need a decent number of samples (100 or more) in the 0.001 quantile space to estimate the profit/loss value accurately and also in finding out what lead to such bad profits by analyzing the samples. One approach for obtaining the samples is to generate large number of
DB instances using MCDB and then select only those which fall in the region of interest of the distribution. This method is not efficient if the probability we are looking at is too small. MCDB-R solves this efficiency problem by adapting Gibbs cloning approach to the MCDB system. Gibbs cloning is an iterative process that moves samples closer to the tail of the distribution function. Each step deletes the uninteresting samples and replace them with clones of interesting samples. This creates larger number of samples, all in the new region closer the the target tail. The new set of samples will be highly correlated due to the cloning. These samples are randomly perturbed such that the new samples after the perturbation are reasonably uncorrelated. During perturbation we need to make sure that the new samples are in the new region we obtained after deleting uninteresting samples. This perturbation is done by Gibbs sampler due to its ability to perform multidimensional samples while satisfying the constraints of the new region.

![Figure 2-7. Gibbs cloning: Clone the elite samples](image)

This Gibbs cloning process is explained through figures. Figure 2-5 shows the the query-result distribution with four initial samples, denoted by the small yellow circles. Assume that our target is to gather samples from the right side of the distribution function. Next step is to delete half the samples which are farthest from the right tail, as shown in Figure 2-6. In this figure, the vertical line shows the boundary of new region.
we created. This line acts as a boundary for any perturbation we perform later. In the next step we clone the remaining two ‘elite’ samples, to create four samples in total. We can see the cloned samples as the green circles in Figure 2-7. We use Gibbs sampler to perform the perturbation while satisfying the constraint. The new four samples after the perturbation are shown in Figure 2-8. Note that all the new samples are after the vertical line, which is our constraint. This process will be repeated as many times as required to move to the tail as required by the query. The process of Gibbs cloning, along with the Gibbs sampler are explained in next section.

2.1 Background

2.1.1 Rare Event Simulation

Rare events are events that occur with very low probability but they could have extreme impact when they do occur. Generating samples in the rare event space for risk analysis is useful in many applications. Some applications which benefit from this are in financial domain, and network reliability \[10, 53\]. Using basic Monte Carlo for rare event simulation is not efficient. When the probability of rare event space is $p$, to obtain a single sample in that space requires $1/p$ basic Monte Carlo simulations. If $p$ is $10^{-6}$, which is possible in many applications \[53\], getting 100 samples of interest requires $10^8$ samples in the full event space, which is clearly inefficient. Two popular techniques
used for this simulation are Importance sampling and cloning [10, 11, 54]. Importance sampling is not good for high-dimensional data because of the problem of numerical instability [10]. Also in this application the dimensionality is typically equal to the number of tuples in the relation, which renders importance sampling useless.

**Cloning:** We view a random database $D$ as a vector of random variables over a very high-dimensional joint distribution. $D$ consists of multiple DB instances $D_1, D_2, \ldots$ each representing a sample from the join distribution. Consider an aggregate query over $D$. *Cloning* algorithm first generates $n_1$ DB instances. At step $i$ it keeps only the top $100 \times p_i\%$ (decided based on the aggregate value of the query over that instance) of the instances and clones these top instances to obtain $n_{i+1}$ new instances. All these are in top $100 \times p_i\%$ but not independent. It randomly perturbs these DB instances within the new tail region (as explained in the next section) to make them independent of each other. The random perturbation process is explained in Section 2.1.2 below. A single step is called a *cloning iteration*, and after each such iteration the algorithm moves further into the tail. The number of DB instances, the number of cloning iterations, and the values of $n_i, p_i$ are dependent on the quantile specification of the query.

### 2.1.2 Gibbs Sampling

The technique above explains how to walk towards the tail after cloning and randomly perturbing DB instances at each step. In this section, we look at Gibbs sampling, which is used for randomly perturbing a given DB instance. Gibbs sampling is a Markov Chain Monte Carlo algorithm for obtaining a sequence of random samples from joint probability distribution. Gibbs sampler requires samples from conditional distributions of each dimension to perform the perturbation of the multi-dimensional sample.

Let $X = (X_1, X_2, \ldots, X_r)$ be an $r$-dimensional random vector. $X$ follows a distribution $h$ and takes values from the discrete set $\chi^r$. Here $h(x_1, \ldots, x_r) = P(X_1 = x_1, \ldots, X_r = x_r)$, for $x = (x_1, \ldots, x_r) \in \chi^r$. As mentioned earlier Gibbs sampler needs to be able to generate
samples from conditional distributions. \( h_i^*(u|v) = P(X_i = u|X_j = v \text{ for } j \neq i) \). \( h_i \) is the conditional distribution for \( i^{th} \) dimension, where \( 1 \leq i \leq r \). Given an initial vector \( X^{(0)} = (X_1^{(0)}, \ldots, X_r^{(0)}) \) the Gibbs sampler generates the random vectors \( X^{(0)}, \ldots, X^{(k)} \). Vector \( X^{(j)} \) is obtained by replacing each of the dimension values in \( X_i^{(j)} \) in \( X^{(j-1)} \) by using \( h_i^* \), where \( 1 \leq j \leq k \).

Consider the perturbation of a DB instance \( D_1 \) from previous section. Treat \( D_1 \) as similar to \( X \). The perturbation process should keep \( D_1 \) in the current tail range seen as a cutoff value on the aggregate, and replace samples in each of its dimensions. In our setting the dimensionality \( r \) is typically the number of tuples in a relation, and dimensions are independent. If we assume such independence then \( h(x) = \prod_{i=1}^{r} h_i(x_i) \), where \( h_i(u) = P(X_i = u) \). \( Q \) be a real valued function on \( X^r \), mimicking an aggregate and \( c \) is the cutoff value for tail and is conditioned over distribution \( h \). Now for perturbing \( i^{th} \) dimension we need a sample from \( h_i(x|c) = P(X = x|Q(X) \geq c) \). We can do this by a Rejection algorithm: keep sampling from \( h_i(x) \) until \( Q(X) \geq c \) is satisfied. For more details see [6].

2.2 Schema Specification

The first step in specifying a query in MCDB-R system is defining a random (or stochastic) relation. A relation is random if it has one or more stochastic attributes. Such attributes are populated through samples from parametric probability distribution functions. These parametric distribution functions are specified through the Variable Generating (VG) functions which are explained in the next section. During the query execution, each stochastic attribute in a tuple is replaced by samples generated from the corresponding VG function. Once the random relations are specified, the next step is writing standard Structured-Query-Language (SQL) for the aggregation. Aggregate operator is specified later, as part of the MCDB-R specific RESULTDISTRIBUTION block. The query compiler then converts these SQL statements into a query plan, which includes both standard relation operators as well as some new operators to generate
and process stochastic attributes. Finally the parameters required for the tail sampling are specified. These parameters determine how far along the tail the final samples are generated. An example is given below for a better understanding of the process.

**Example:** The example query we use here describes a predictive model of customer demand. This query simulates the effect of a 5% increase in price on the final profits. For calculating the new profit we need the new customer demand due to increased price. The demand is estimated using Bayesian model on the existing data. A common prior distribution for demand is used for all customers. The the posterior distribution is calculated based on the parameters specific to customer. The VG function outputs the possible customer demand values due to changes in price. This query requires the specification of the random relation `demands` as given below.

1. CREATE TABLE demands (new_dmnd, old_dmnd, old_prc, new_prc, nd_partkey, nd_suppkey) AS
2. FOR EACH l IN (SELECT * FROM lineitem, orders WHERE l_orderkey=o_orderkey AND yr(o_orderdate)=1995)
3. WITH new_dmnd AS Bayesian (SELECT p0shape, p0scale, d0shape, d0scale FROM params WHERE l_partkey = p_partkey)
4. (VALUES (l_quantity, l_extendedprice*(1.0-l_discount))/l_quantity, l_extendedprice*1.05*(1.0-l_discount)/l_quantity))
5. SELECT nd.value, l_quantity, l_extendedprice*(1.0-l_discount)/l_quantity, 1.05*l_extendedprice*(1.0-l_discount)/l_quantity, l_partkey, l_suppkey
A random relation is generated on the fly. Therefore, the above statement needs to specify how each of the attribute is derived, either from base tables in case of deterministic attributes or form the parametric VG function in case of stochastic attributes. Lines 1 and 2 give the table name along with the list of attributes. The sub-query from lines 3 to 5 creates the base table, and each tuple in it will contribute to a set of tuples in the random relation demands. The stochastic attribute new_dmnd is generated from the distribution function specified from lines 6 to 12. The name of the VG function used for sample generation is specified (as Bayesian) on line 6. The table params has the parameters required for the Bayesian VG function. The params table can have one or more tuples for each matched tuple in the base table. The join condition between these two tables is given at line 9. In the predicate, 1_partkey is from the base table and p_partkey is part of the params table. Some parameters to the VG function are based on the base relation. These are specified after the keyword VALUES at the beginning of line 10. The relation new_dmnd(value) is created from this entire VG function block (lines 6 to 12). The attributes in the relation demands are populated according to the SQL statement in lines 13 to 17. In this example, the first attribute new_dmnd.value is stochastic, and rest are deterministic. All the deterministic attributes are calculated as functions of attributes in the base table.

18. SELECT SUM (new_prf-old_prf) AS totalProfit
19. FROM ( 
20. SELECT 
21. new_dmnd*(new_prc-ps_supplycost) AS new_prf 
22. old_dmnd*(old_prc-ps_supplycost) AS old_prf 
23. FROM partsupp, demands 
24. WHERE ps_partkey=nd_partkey AND 
25. ps_suppkey=nd_suppkey)
26. WITH RESULTDISTRIBUTION MONTECARLO(100)
27. DOMAIN totalProfit >= QUANTILE(0.001)
28. FREQUENCYTABLE totalProfit

Writing the aggregate part of the query is quite straightforward and is given from line 18 to 25. Note that the stochastic attribute `new_dmd` is part of the aggregate. Information to the tail sampling operator is given from line 26 to 28. The value `totalProfit` is the aggregate calculated in this query. The quantile of the final samples in the query-result distribution is specified with QUANTILE keyword. For this query the samples are in the $10^{-3}$th quantile. The number of samples generated with that quantile is given with MONTECARLO keyword in the line 26, which is 100 in this query. The direction of the tail is given by $\leq$. In this query, the samples are to be generated in the lower tail of the query-result distribution. The FREQUENCYTABLE keyword specifies where the samples generated by the query are stored. A table `FTABLE(totalProfit, FRAC)` is created, where in each tuple `totalProfit` is a distinct sample value and the corresponding `FRAC` is fraction of the 100 samples in which that value was observed.

2.3 The Variable Generating Function

The VG function is one of the vital components of this system. It takes in parameters and pseudo-random number generator (PRNG) seeds and returns the samples, which are then stored in the stochastic attributes. The framework surrounding the VG functions allows the users to provide their own functions along with the process in which that function receives the parameters and returns the samples. A standard set of VG functions like `Normal()` or `Poisson()` are already provided with the system. The framework is very flexible and allows specification of a variety of VG functions including multi dimensional distribution functions with correlations.

**Framework:** As mentioned above, the VG function framework is designed to allow the user to create his own function. The Bayesian VG function mentioned in the previous section is fairly complicated and is not part of the standard library functions.
Any custom VG function like that can be added to the library. Any VG function should implement the following five methods – Initialize(), TakeParams(), OutputVals(), PrepareNextTrial() and Finalize().

- **Initialize()** takes in an array of PRNG seeds. These are the seeds that the VG function uses for pseudo-random number generator. The size of this list should be same as the number of samples to be generated. This is because, in MCDB-R, each sample has its own seed. So each of the independent and identically distributed samples generated has its own series of pseudo-random numbers, and can be generated independent of its previous samples. This is vital because MCDB-R can require samples to be generated out of sequence. Refer to Section 2.7 for more on why we need samples out of their sequence. This method also initializes the pseudo-random number generator using the first of the seeds and any data structures and variables required by the VG function.

- **TakeParams()** is used to send any parameters required by the probability distribution function used in the VG function. The parameter values generally vary per tuple, and this information is specified by the **FOR EACH** clause in the **CREATE TABLE** statement. Each tuple may need multiple calls to **TakeParams()** depending on the VG function.

- **OutputVals()** returns the samples from the VG function. These samples can be multi-dimensional, therefore for each sample a sequence of calls are to be made to the **OutputVals()**. The method returns a **NULL** after it returns the value for the final dimension of the sample. Each non **NULL** return value results in a separate tuple for the sample. So a single input tuple can result in multiple output tuples.

- **PrepareNextTrial()** is called at the end of each sample – i.e. after each **NULL** return value from an **OutputVals()** call. This method will initialize the pseudo-random number generator to the next PRNG seed.

- **Finalize()** is called at the end, after all the samples are generated. This method will de-allocate any data structures created by the VG function and the seed array. After a call to **Finalize()**, the VG function object is fully reset and is ready for an **Initialize()** call from the next tuple.

Consider the Bayesian VG function in the query from previous section. The parameters list sent to **TakeParams()** has the values \( (p0shape, p0scale, d0shape, d0scale) \). A single call to **TakeParams()** is sufficient because the query has a single parameter table and one tuple each for the parameters. If the VG function were a multi variate probability distribution, sending the co-variance matrix would have required multiple
calls to `TakeParams()`. A call to `OutputVals()` will again return only a single value, the `new_dmnd.value`. A second call to `OutputVals()` will return a `NULL`, and the subsequent call to `PrepareNextTrial()` will advance the PRNG seed.

Note that this framework only specifies how the parameters and PRNG seeds are sent. The process of sending the correct parameters and collecting the returned samples to form the final tuple is done outside the VG function. This process works as a wrapper around the VG framework and is described in the Section 2.5 as part of the `Instantiate` operator.

### 2.4 The Seed Operator

This operator adds `Seed` attributes to random relations. For each VG function in the `CREATE TABLE` statement of a random relation, one `Seed` attribute is added. The `Seed` attribute consists of a seed identifier and the current set of active samples. The seed identifier must be unique over all the random relations in a query. This identifier is used as part of all the PRNG seeds created for this tuple. So multiple tuples with same Seed identifier will create unwanted correlations in the final result.

![Seed attribute example](image)

Figure 2-9. An example Seed attribute

Along with the seed identifier the attribute also stores the number of DB instances for this query, and a list of the current iterations assigned (called active iterations) for each of these DB instances. The number of DB instances is same as the number of samples asked in the query with `MONTECARLO` keyword. The need for the list of these iterations is explained later in Section 2.6.1. Figure 2-9 shows a `Seed` attribute. The attribute has 1 as seed identifier, and the number of DB instances are 4. The third
value points to the list of active iterations. The list has one value for each of the 4 DB instances. The iteration number assigned for DB instance 1 (DB1) is stored in DB1 iter. cell.

### 2.5 The Instantiate operator

Instantiate is a fundamental operator in MCDB-R. This operations acts as a wrapper around the VG function. It takes the random relation, and a series of parameter relations as input. The parameters from inner relations are matched with the corresponding tuple in the outer relation through a join predicate. For each tuple from the outer relation, a series of calls to the VG function are made. These calls send the information required by the VG function including the PRNG seed, parameters and an identifier. Once the VG function returns the samples, they are merged with the corresponding tuple in the random relation, to form the final tuple.

**Example:** Let $\text{orders}\_\text{rnd}(\text{orderkey}, \text{year}, \text{total}\ AS\ \text{Normal}(\mu, \sigma))$ be a random relation (see query Q2 in Appendix). This relation is generated by passing $\text{order}(\text{orderkey}, \text{year}, \text{seed})$ as the main input table, and $\text{params}(\text{orderkey}, \mu, \sigma)$ as the parameter table. Instantiate matches the parameters with corresponding tuple in the order based on the join on orderkey. The attribute order.seed contains the base seed used to calculate the PRNG seed array.

The Instantiate operator generates the samples from a parametric VG function. For each random tuple Instantiate receives name of the VG function to be used, the parameters to it and the locations to store the generated samples. Parameters can be passed directly through the main "outer" input, or through multiple parameter "inner" inputs. See [37] for more details.

Inner working of this operator is shown in Figure 2-10 for one input parameter table. If we consider the example from last paragraph, the inner input will be the table params. The outer input is the order relation. Once Instantiate gets one tuple from the outer relation, it makes multiple copies of the tuple, one per each inner input pipes.
These copies are used to join with the parameter tables based on the key attribute. Through this join, all the parameters related to the key can be brought together by the sort operation just after the joins. Remember from previous section that a seed identifier is unique to each key in the outer input relation. Therefore, sorting on seed identifier will bring together all the parameter tuples and the outer input tuple. Sorting is not done on the key because it is not necessary any more in the Instantiate and can be dropped and seed will be the new identifier for the tuples.

After sorting on the seed identifiers is finished, the parameters required for the VG function are brought together. The parameters can now be rearranged as required and sent to the VG function. In the above example, the $(\mu, \sigma, \text{seed})$ are sent to the
VG function. Once the VG function generates the sample array it is returned to the Instantiate and it is then merged with a copy of outer input tuple. The outer input pipe is also sorted on seed identifier, therefore the merging process is straightforward. The merge function will push the resulting tuples into the output pipe going to the next operator. Note that many VG functions would need more than one parameter table, and more than one tuple from a parameter table can join with a single key from outer input relation. One extreme example for this is a multivariate distribution function. This function requires the covariance matrix to be passed to the VG function. One way to pass this matrix is through the sparse matrix representation. The Instantiate operator and VG function framework are very flexible and designed to allow all kinds of complex input methods.

Instantiate is one of the expensive operators in this system. One reason for this is due to the joins and sorts on inner and outer tables. Since this operator also encapsulates the VG function, if the sample generation is expensive then that will again slow down the operator. The other major contributor to the expense here is the memory allocation for the sample array. The sample arrays are generally very large, hence even the data movement between VG function and the array is considerable.

### 2.6 Gibbs Tuple Bundles

For generating $num_{db}$ (number of DB instances) samples in the tail, we need to perform cloning and perturbation on initial DB instances. To keep track of the current samples used, we maintain the DB instance information in the tuple, stored as part of the Seed attribute. A stochastic attribute is a stream of random samples, and only $num_{db}$ of them are in use at any point in time (assigned to $num_{db}$ DB instances). The size of the sample stream is denoted by $num_s$. At the perturbation step we do not know which of the samples will be used in the new DB instances. So we carry the whole sample stream without any filtering (due to selection predicates) on stochastic attributes. The filtering is performed once the sample is assigned to a DB instance. The stochastic table
Tail Sampler():
1. Parameters:
2. \( n \): number of cloning iterations
3. \( num_{db} \): number of DB instances
4. \( num_{cloned} \): number of DB instances cloned at the end of each iteration
5. \( num_{s} \): size of the sample stream
6. Variables:
7. \( c \): cutoff value for the aggregate
8. PQ: priority queue on seed identifiers
9. R: input relation with Gibbs tuples
10. r: current Gibbs tuples with same seed identifier
11. InitializeDB(R, \( num_{db} \))
12. PQ(R)
13. for each iteration \( i \) from 1 to \( n \)
14. Clone(PQ, \( num_{db}, num_{cloned}, c \))
15. PQ.reset()
16. while(\( r = PQ.nextSeed() \))
17. for each DB instance \( j \) from 1 to \( num_{db} \)
18. while(\( r[j].Advance() \))
19. if Agg(\( r[j] \)) \( \geq c \)
20. found = true
21. end if
22. end while
23. if not found
24. \( num_{s} = num_{s} \times 2; \)
25. QueryRerun()
26. end if
27. end for
28. end while
29. end for

Figure 2-11. Tail sampler operator

CustOrder from Figure 2-3 with tuple bundles and seed is shown in Figure 2-12. In this relation NEW QTY is the stochastic attribute. The multiple DB instances on the left side in Figure 2-12 is the logical representation, where as the right side STOC. CustOrders is the storage representation in MCDB-R.

We run the query plan multiple times to replenish the sample stream whenever we run out of the samples. Due to operations like stochastic join some new tuples can
Figure 2-12. CustOrder with Gibbs tuples

be added and old tuples eliminated from the new run. So there is a need to maintain persistent DB instance information outside a query plan execution. So for each unique Seed in the Gibbs tuples we have a Tail-Sampling-seed (TSSeed) object on disk. Seed and its corresponding TSSeed are synched whenever a modification is performed to Seed. A TSSeed object stores the seed identifier, PRNG seed and the DB instance information. It also stores the current in-use range of the sample stream and starting of the unused samples.

2.6.1 Tail-Sampling-Seed and Seed Queue

Figure 2-13 shows an example TSSeed object and corresponding Seed object in the tuple. Each TSSeed object stores the seed identifier for the tuple. This identifier is used to match the TSSeed objects and the related tuples in the query pipeline. A TSSeed stores the current range of generated samples, and the current assigned range. It also maintains the current assigned iteration number for each of the DB instances. The TSSeed class has the public function `GetNumSeeds()`.
returns a list of PRNG seeds used later in VG function for seeding the pseudo-random number generator. Each of the PRNG seeds returned is unique, and is calculated through a hash function on the seed identifier and the iteration number. TSSeed object provides the method `Advance()` to move the current iteration number to next previously unused iteration number. This method is used in the Gibbs sampler to replace the current sample with a new one. `Advance()` returns an error value if the sample array is exhausted.

All the changes to assigned iteration numbers are done through the TSSeed, and the changes are reflected onto the Seed through a `Synch()` function. The Seed object also stores the current assigned iteration numbers. But these iterations numbers are relative to the positions in the sample array (the stochastic attributes). The `Synch()` transforms the absolute iterations numbers in a TSSeed object into the relative iterations numbers in the Seed object.

### 2.6.2 Split Operator

<table>
<thead>
<tr>
<th>ID</th>
<th>STOC. ARRAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 1 1 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JOIN ATTR</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>67</td>
</tr>
<tr>
<td>2</td>
<td>85</td>
</tr>
</tbody>
</table>

Figure 2-14. Stochastic table in a join
A stochastic array attribute stores multiple instances of the attribute, hence a fixed order can not be defined on this attribute. So operators like the join, duplicate elimination, multi-set union, intersection, and difference can not be applied on stochastic array attributes directly. A Split operator is used internally in each of these operators to break up a stochastic attribute with an array of samples into a set of smaller attributes each with one unique value in the sample array. A bit string is used to store the locations of that value in the sample array. The new attribute with a single sample is called a stochastic single attribute. A bit string is always stored along with the stochastic single attribute.

Consider the two tables in Figure 2-14. The right table has a stochastic attribute. To simplify the example, the Seed attribute is not shown in the right table. The join predicate is on STOC. ARRAY and the JOIN ATTR attributes. Figure 2-15 shows the left table after the split. Now it is straightforward to match the tuples on the join predicate. The result tuples after the join predicate are shown in Figure 2-16. The attribute STOC. ARRAY is broken up and is renamed as STOC. SINGLE. The new attribute BITSTRING keeps track of the occurrence of the stochastic single value in the stochastic array. Note that the appearance of these tuples in the join result depends on weather the iterations
numbers of the active bits in `BITSTRING` are currently assigned to a DB instance. If only the iteration number 0 is active, then the first tuple in Figure 2-16 with stochastic single value 1 will not appear in the result, where as the second tuple with value 2 will appear. If an `Advance()` call is made, then the active iteration number is 2. In this case, tuple 2 will not appear and tuple 1 will appear in the result.

2.7 Gibbs Looper and Tail Sampler

Gibbs looper takes as input the number of DB instances to be generated `num_{db}`, and the quantile to be estimated. It then calculates the number of cloning iterations and the number of DB instances to be cloned after each iteration. Figure 2-11 gives the Tail sampler method. This method performs the necessary cloning iterations on the given input Gibbs tuples. We need to operate on all Gibbs tuples with same seed identifier together, so a priority queue PQ is used for that purpose. At the beginning of each iteration we clone (line 14) the top DB instances. The while loop at line 16 goes through the Gibbs tuples and performs the perturb operation. All Gibbs tuples with same seed identifier are returned by PQ.nextSeed() call. Now the samples in each DB instance of this set of tuples \( r \) are replaced (lines 17-27). The while loop from lines 18-22 performs the rejection algorithm. After an `Advance()` the new sample is accepted only if the new aggregate value is greater than (or smaller than) the cutoff.

2.7.1 Gibbs Queue

Gibbs queue is used by the Gibbs looper to retrieve the records in the seed order. Gibbs queue is a disk based priority queue which allows online insertions while keeping the sort order. In other words it allows removal of records and also at the same time insertion of new records. Gibbs queue is designed such that interleaved remove() and insert() operations are performed with good efficiency. Figure 2-17 shows the mechanism of the Gibbs queue. The first phase is similar to a disk based two phase merge sort [26]. A series of sorted runs are created on the disk. A primary priority queue is set up to merge records from the runs. For disk efficiency we employ page
buffering. Gibbs queue also has another in memory (secondary) priority queue, which is used to hold the reinserted tuples. The top tuple can now can be in either primary or the secondary queue. This mechanism keeps the Gibbs queue ordered even after reinserts. The size of the secondary queue is as big as the disk based run. Whenever this queue is full, a new run is created on the disk. The new run is also attached to the primary queue. The secondary queue will now be empty and is ready to receive new tuples that are reinserted. This functionality of reinserting a tuple is very important for the Gibbs looper, since there is a possibility that a Gibbs tuple can have more than one seed. In such a case the tuple should be processed at Gibbs looper once per each seed. How to process such tuples is explained in more detail below.

**Preparing tuple bundles with multiple seed identifiers:** Gibbs queue holds the tuple bundles in sorted order according to the seed values. But it is possible that each tuple bundle can have more than one seed. This can happen due to two reasons: (1) the
A tuple can have more than one random attributes, generated by different VG functions, and hence has one seed for each VG function. (2) due to merging of tuples from two relations, each with random attributes, inside a Join operator. The tuple bundles with more than one seed needs to be processed by Gibbs looper once per seed. For this, the Gibbs looper should be informed that the tuple has more seeds, and needs to be reinserted into the queue. A function `Advance()` is provided by the tuple for this. The tuple bundle also provides two other functions to allow this iteration over seeds: `BuildSeeds()`, and `ResetSeeds()`. `BuildSeeds()` is the initialization function, which collects all seeds in the tuple, sorts them and stores them in an array in ascending order. It also initializes an index on the array pointing to the least valued seed. An `Advance()` moves the index to next seed. `Advance()` returns false if the array is exhausted. The `ResetSeeds()` moves the index back to the first seed. The Gibbs looper after processing a tuple, will `Advance()` the seed. If the tuple has more seeds, then it will reinserts the tuple into the Gibbs queue.

Figure 2-18. Example: End of run 1

### 2.7.2 Rerunning the Query Plan

The query plan is rerun if we are out of samples for a given seed identifier before finding a suitable replacement for all the DB instances. For each successive run the number of samples generated for each seed identifier is doubled, provided the record fits inside the page. Once the tail sampler decided to rerun the query, it will delete all the existing tuples, and flush all TSSeeds and reset the seed queue. If with in page size
Figure 2-19. Example: Beginning of run 2

limits, the size of samples is doubled. Once this is done, tail sampler exits to give the control to the query planner. Once the planner sees that the required number of Gibbs loops are not finished, it will restart the query plan. At Instantiate the new sample stream is started only from the last used sample from the previous run. Figures 2-18 and 2-19 show the TSSeed and Seed at the end of first run and at the beginning of second run respectively. In Figure 2-18 the range of sample iterations is from \textit{low iter.num}. 0 to \textit{high iter.num}. 5. It can be seen that the samples are exhausted in this array since the \textit{max.used iter.} is 5, same as \textit{high iter.num}. The current assigned iterations are same for both TSSeed and Seed objects since the starting iteration number in TSSeed is 0. For this example consider that the size of sample array for the second run remain as 6. In Figure 2-19 the \textit{lower iter.num}. in second run will start at 2, since that is the largest used iteration number in the previous run (for DB instance 3). The \textit{max.used iter.} will not change and the \textit{high iter.num.} will be 7 (\textit{low iter.num.} + 6 – 1). After the \textit{high iter.num.} is updated, the TSSeed object is synched with Seed object. The current assigned iterations numbers in the Seed will change because the are relative to the \textit{low iter.num.} in the TSSeed.

2.7.3 Materializing the Operators

In the first execution of the query plan the results of any subtree that is totally deterministic are saved to disk. The operator at the root of that subtree is marked as materialized, and in the next query run the results are directly read from the disk.
Figures 2-20 and 2-21 show the plan for query Q1 (see Appendix) before and after materialization process. In the query plan in Figure 2-20, the subtrees at Seed and Group-By operators are totally deterministic. Both the subtrees can be materialized at those operators. The next operator in the query plan is Instantiate, which is definitely not deterministic. So the materialization can not get past Instantiate. Figure 2-21 shows the
query plan after materialization. Figure 2-22 show the first three runs of the execution. The first run is the whole query plan, and second run onwards the modified plan with materialization is used.

Figure 2-22. Query plan with multiple reruns
CHAPTER 3
RELATED WORK

The work presented in this dissertation is very specific to MCDB-R, and therefore is difficult to put in the context of previous research (the related work for the problem discussed in Chapter 4 is given in Section 4.6). In this chapter we cover the research loosely related to MCDB-R. Other works about integrating statistical operations with a relational database are explained in Section 3.1. Monte Carlo methods are used quite heavily in fields such as probabilistic databases (see Section 3.2), and data mining and machine learning (see Section 3.4). In database research, the area closest to MCDB-R functionality is top-k ranking in probabilistic databases. Probabilistic top-k is described in Section 3.3. State of the art risk analysis softwares are explained in Section 3.5.

3.1 Statistical Operations in Relational Databases

MCDB-R is closely related to the idea of integrating statistical operations with relational databases such as MauveDB [21], FunctionDB [63], and SciDB [59]. MauveDB integrates statistical models into a database, and provides the user with model-based views. In that system a user can specify, create and query these views. FunctionDB provides support for continuous functions inside the database, and SciDB aims to provide advanced analytical capabilities inside the databases.

This system is also related to conditioning in Probabilistic Database Systems [41]. Conditioning is aimed at analytical calculation of the confidence values of tuples in the query result. It transforms a given probabilistic database of priors to a probabilistic database of posteriors. Now the confidence values of the tuples in a query result can be calculated for the posterior database. MCDB-R is different as it is based on Monte Carlo based approximations, instead of calculating exact confidence values.

3.2 Probabilistic Databases

Probabilistic databases are used to store data uncertainty in relational model. They have been studied for a long time. Interest in them is due to the applications like
sensors, data integration and cleaning. Methods like information extraction generate uncertain data [31], that can be managed by a probabilistic database. Probabilistic databases can also be used for what-if analysis [37] where the future data is modeled as probability distributions. Processing probabilistic data is studied extensively in recent times; Trio [69], MayBMS [5], MystiQ [50], Orion [13], MCDB [37], PrDB [56], and BayesStore [67] are some of the recently developed probabilistic databases. For a recent survey of this field see [61].

Modeling uncertainty in these databases is done primarily in two ways: tuple level, and attribute level uncertainty. In tuple level uncertainty, a tuple has a probability of occurrence in the relation. It can either occur or not. In attribute level uncertainty, the attribute value is a random variable. Some of the early work on probabilistic databases is done in [34]. They define v-tables, which has variables that represent incomplete information. In v-tables, many of the relational operators can be run on them similarly to that on a relational table. The v-tables can not support projection and arbitrary selections. They also define a conditional table (or c-table), a variation of v-table with a condition for each tuple. This framework does not handle dependencies between attributes.

[7] uses an attribute level uncertainty model. Their relation has a deterministic key and both probabilistic and deterministic attribute groups. Their probabilistic attributes represent a discrete probability distribution. They show that projection and selection are loss less. Some new operators specific to the probabilistic model are also given. For example, x-select operator works by matching probability distributions that are close. [24] presents probabilistic relational algebra, and the tuples are assigned probability values. These values give the probability that the tuple belong to the relation. They use intentional semantics to evaluate the query. In intentional semantics, each tuple has an associated event. After the query evaluation each output tuple will have an expression over the events from which that tuple is derived from (lineage of the output tuple). This
expression can be very complex. Then they need to solve this (sometimes complex) expression to find the probability of the tuple in the output relation. [42] stores probability intervals along with attributes in their probabilistic database system, Probview. They define multiple relational operators like projection, selection, and cartesian product on their data representation, along with a new operator called compaction. Compaction works like a duplicate removal by combining the probability intervals of tuples with same values. They also gives an incremental algorithm to maintain views on their database for insertion and deletion operations.

In contrast to intensional query evaluation in [24], [18, 50] uses extensional query evaluation. In extensional evaluation, the operators calculate the actual probability value even at the intermediate stages of the query plan. This avoids solving the complex expression that we saw in intensional semantics at the end of the processing. [18] shows that some query plans give incorrect results with extensional evaluation. So they give a method to find a safe extensional query plan (by query rewriting) when it is possible to do so. For unsafe plans, they describe a Monte Carlo based simulation algorithm, along with an approximation algorithm. The complexity of executing queries in probabilistic databases is explored in [19]. They show that the complexity of evaluating a conjunctive query over a probabilistic database with tuple independence assumption is either PTIME (Polynomial TIME) or #P-hard. They also give an algorithm that recognizes whether a query is safe or not. Trio [2, 8, 69] keeps lineage information along with the uncertain data. In their model, probability of occurrence is stored in the tuples, and is called the confidence value. Some relational operators like projections are not closed on this data. But lineage is useful in calculating the new confidence values for a tuple after the operation. The query model in [8] allows the user to specify the minimum confidence value required for the tuples to be in the output (Top-k queries). They use lineage to calculate confidence values over the query results on uncertain data. This method is related to the intensional semantics, and the lineage expression
sometimes can become very complex. In such a case a Monte Carlo approximation method [40] is used to solve the expression. [4, 5, 33] introduces U-relations to capture uncertainty in the data. A U-relation has random variables whose probability values are defined in another table. They achieve attribute level uncertainty through this representation. They too use intentional evaluation model on their U-relations. The difference with previous approaches like Trio is that they take less space to store the uncertain data. To calculate the confidence values of output tuples (if the expression is too complex) they use Monte Carlo approximations [40].

MCDB [37] uses Monte Carlo sampling to estimate (user defined) statistical models on uncertain attributes in a relational database. MCDB treats the database with uncertain attributes as a joint probability distribution. Since the dimensionality of that distribution is very high, Monte Carlo sampling is used for estimating query results over that joint distribution. Multiple database instances are created through sampling. Each uncertain attribute is defined by a probability distribution and the attribute value is represented by a sample from that distribution. Each of the database instances are generated independently through a pseudo-random generator. For efficiency MCDB bundles all the database instances of a tuple together, and executes the query over that tuple bundle only once for all instances, instead of once per instance. MCDB’s framework of user defined statistical models allows flexibility in the types of models and the types of queries. PrDB [20, 56] uses probabilistic graphical models as base models. This gives PrDB the flexibility of representing tuple level uncertainly, attribute level uncertainty, as well as correlations with in attributes, and with in tuples.

3.3 Top-k Ranking in Probabilistic Databases

Top-k ranking in probabilistic databases, although recent, is well studied [9, 15, 28, 32, 44, 49, 58, 65, 72]. There have been multiple definitions of Top-k tuples in a probabilistic database due to the existence of confidence (probability of occurrence) along with the actual value (or score). MCDB-R differs from this work as it generates
values after a given quantile. Though the top-k ranking can be used to generate values
with low probability they not suitable for restricting the values within a given quantile.

[58] first studied this problem, by giving two definitions of top-k tuples: Uncertain
Top-k and Uncertain Rank-k. An Uncertain Top-k query returns a list of k tuples,
which appears as the top-k answer in most possible worlds. For Uncertain Rank-k,
each tuple returned for a position should be occurring at the position in most possible
worlds. That is, for $i_{th}$ rank, we return a tuple that occurs at $i_{th}$ rank in most possible
worlds. The problem is formulated as a state space search. The algorithms scans the
probable tuples in top-k and maintains a large number of states. Each state is a set of
possible worlds that have common top tuples. [72] gives more efficient algorithms for
Uncertain Top-k and Uncertain Rank-k. They use a representation called x-tuples
in their algorithms. Each x-tuple has multiple options, each option has an associated
probability. These probabilities define a discrete distribution over all the options for that
tuple. The algorithm uses a hash map to store the x-tuples, such that if an id of a tuple is
given then its confidence value can be retrieved fast.

[32] gives a probabilistic threshold top-k query. A tuple will be in top k if it has
at least p probability to be in the top-k lists in all possible worlds. [32] gives an exact
algorithm to compute this, along with some pruning techniques using the probability
threshold to improve the the efficiency. A sampling based approximate method is also
given. [15] defines expected ranks in both attribute level and tuple level uncertainly
models. The rank of a tuple t in a possible world is the number of tuples whose
score/value is higher than t. They give exact algorithms and some pruning methods
to calculate expected rank based top tuples, both in attribute level and tuple level
uncertainty models.

Finding nearest neighbor probabilistic top-k objects is studied in [9]. They proposed
approaches based on input/output (I/O) and Central Processing Unit (CPU) cost.
Their query processing algorithms are designed to minimize the total cost. [28]
gives a method gives some emphasis to scores in the top-k selection. They define the c-typical top-k, which returns c typical top-k vectors, according to the score distribution (A typical set in information theory is a set of sequences whose probability is close to $2^{-e}$, where e is the entropy of their source distribution). There is a good chance that the actual top-k score is close to one of these c vectors.

In [49] top-k is defined only on probability values. K tuples with highest probability are given in the output. For efficiency, they run several Monte Carlo simulations together one for each candidate in the result set, and calculate the probability values to minimum accuracy required. A recent work [43, 44] provides a unified framework for learning multiple ranking functions over a probabilistic database. A set of key features are identified, and parameterized ranking functions are defined over these features. The parameters are chosen based on the user preferences. These parameterized functions can encompass many of the top-k methods defined above. [44] presents several efficient algorithms to calculate these ranking functions.

### 3.4 Monte Carlo Methods in Databases, Data Mining, and Machine Learning

The specific Markov Chain Monte Carlo (MCMC) based Gibbs cloning (or splitting) idea [10] adapted in MCDB-R is mainly developed for rare event simulation. [14] presents novel algorithm for sensor deployment based on splitting and Gibbs sampler approach from [10]. Their focus is to find the best spatio temporal placement of the sensors to improve the detection of a target that is intelligent, reactive and moving with a strategy to go undetected. This is an optimization problem, and genetic algorithms are used previously to solve it.

One of MCMC’s popular uses is in machine learning, as part of the Bayesian inference framework. A good introduction to using MCMC for Bayesian inference is given in [25]. MCMC is used as part of inference framework for modeling dynamic user interests in [3]. For scalability for millions of users, their inference procedure runs incrementally a fast batch Gibbs sampler on the data at a given time instance, give
the state of sampler at previous time instances. Gibbs sampling and latent Dirichlet Allocation (LDA) have been popular in learning Topic models from text corpora since it is, in theory, more accurate than the variational methods. Topic modeling on text documents is a popular area since it has applications among others in information retrieval and document clustering. The drawback of Gibbs sampling for this application is its efficiency, and hence scalability for large datasets. A novel and faster method of collapsed Gibbs sampling for LDA is proposed in [47]. Collapsed Gibbs sampler integrates out some variables and sampling is done over the simplified distribution with only the specific variables of interest. [64] used collapsed Gibbs sampler for document clustering. The model used is a mixture of multinomials, with Dirichlet priors. Through experiments, they show that using MCMC is better than using Expectation-Maximization algorithm for their model. Various methods for topic modeling are compared in [71]. Some of those methods were based on Gibbs sampling, and some are variational inference methods like maximum entropy. They show that their new sparseLDA is faster than previous LDA based methods. Their performance improvement is due to a new data structure that results in fast sampling even with a low memory usage. Scaling of MCMC methods for larger datasets is studied both in machine learning [12, 23, 57] and databases [66, 73].

Basic Monte Carlo method also is previously used for query processing in probabilistic databases [27, 37, 39, 68]. Processing queries on array databases require Monte Carlo simulations [27]. This requirement is because of the complexity of array and vector operations required by the applications in science and engineering. Using exact method is definitely infeasible due to the dimensionality, and methods like Gaussian mixture models can not produce closed form results other than + and - operators. A second reason is that the models used to represent correlated uncertain data in array data, like graphical models, require Monte Carlo approximate calculations. In [27], authors use Markov Random Fields as model and give solutions
to complex operators like array join. MCDB [37] uses Monte Carlo sampling to estimate (user defined) statistical models on uncertain attributes in a relational database. It provides samples from the query-result distribution, instead of a single value as output. This paper is discussed in more detail in Section 3.2. Monte Carlo methods are used in [39] for evaluating lineage processing on correlated probabilistic data. Specifically, Gibbs sampler is used in this setting for approximate evaluation of conjunctive queries with large complexity. While exact evaluation of lineage processing on tuple independent model is polynomial time, complexity for for that in correlated tuples is #P complete [39]. In [68] factor graphs (a type of probabilistic graphical model) are used to represent uncertainty over relational data. To avoid the #P complexity of query evaluation, [68] uses MCMC based inference. The specific MCMC algorithm they used is Metropolis-Hastings. They propose a new algorithm combining Metropolis-Hastings and materialized view maintenance. View maintenance technique is used for performance improvement.

In [22], MCMC is used for data cleaning. Their dataset consists of facts collected through crowdsourcing, and their main goals are to identify which of the facts are reliable, how to answer questions based on the collected data, and what questions to present next to the users to improve the data quality. The dataset is then treated as a probabilistic database and some probabilistic and recursive rules are defined for interpreting the data. As we have seen before, evaluating queries exactly on probabilistic databases is infeasible. MCMC based algorithm for approximate calculation of the query result can be used. But MCMC is guaranteed to produce good approximations only if the data cleaning rule set is strongly connected. A disconnected rule set is not ergodic, a required property for good MCMC approximations. [22] uses MCMC separately on each connected component of the rule set and combines the results.

Monte Carlo methods, apart from being used to evaluate queries over probabilistic databases, are also used to create the probabilistic databases from uncertain or missing
data. An inference algorithm based on Gibbs sampling is proposed in [60] for predicting the probability distribution over multiple missing values. They use this algorithm as part of their learning framework to create probabilistic databases. A joint distribution over the data is approximated with a set of conditional probability distributions, that together are called a dependency network. Each of the conditional probability distributions is learned independently, and hence can not guarantee a consistent joint distribution. Gibbs sampling inference techniques are used to recover an approximation of the joint probability distribution even though the individual conditional distributions are inconsistent.

### 3.5 Risk Analysis Software

Monte Carlo simulations are implemented in many risk analysis software [16, 45, 62]. MCDB [37] too allows scalable risk analysis with its flexible VG function framework. Oracle Crystal Ball [45] supports predictive modeling, forecasting, Monte Carlo simulation and optimization. It is based on Microsoft Excel, and supports "what-if" risk analysis. @RISK [16] along with an implementation of Monte Carlo simulations, provides RISKOptimizer to speed up the simulations. The software based on spreadsheets like Crystal Ball and @RISK will not be as scalable as MCDB because they lack disk based algorithms, and if an application exceeds random access memory (RAM) size, then most time is spend in swapping data to and from disk. The other stand-alone risk analysis softwares like Analytica [62] are proprietary and it is difficult to tell if the implementations are disk optimized. Analytica claims to be twenty times faster than risk software implemented on standard spreadsheets – so they might be using some disk based algorithms. Note that none of these risk analysis softwares, to our knowledge, support tail sampling.
CHAPTER 4
SERIALIZING SAMPLE GENERATION PROCESS

Figure 4-1. An MCDB-R query plan

4.1 Background

MCDB-R system uses sampling based Monte Carlo simulations to perform risk analysis on uncertain data. In MCDB-R, the data uncertainty is modeled through parametric probability distribution functions. The whole database is seen as a high dimensional joint distribution. Extreme risk analysis on this data can be done by observing the low probability regions – the upper and lower tails of a query result distribution. For a given query, to analytically solve either the result distribution or any estimate on the tail of the result distribution is very difficult due to the high
dimensionality. Therefore, MCDB-R uses Monte Carlo sampling based Gibbs cloning to efficiently generate samples in the tail. By using these samples we can approximately calculate the estimates. The accuracy of these estimates depends on the number of samples. Gibbs cloning method is used in MCDB-R because it is more efficient in generating large number of samples from the tail. Gibbs cloning is an iterative process that moves further into the tail after every iteration. Each iteration has a cloning step and a perturbation step. In the cloning step we delete the samples that are away from the tail and replace them with copies of samples that are further into the tail. The new set of samples will be highly correlated due to cloning, but they will be further into the tail. These correlated samples are randomly perturbed such that the new samples after perturbation are uncorrelated but still remain close to the tail. The distance to the tail is enforced through a constraint on the aggregate value. Perturbation is done by replacing the existing sample with another random sample that still satisfies the constraint. Finding a good replacement for multiple dimensions of the data together is extremely difficult. Hence, the replacement on high dimensional data is done by a Gibbs sampler. Gibbs sampler perturbs the samples from one dimension at a time. At a given dimension the replacement of an existing sample is found through the Rejection algorithm (Section 2.1.2). The Rejection algorithm replaces the current sample with a new random sample and checks the constraint on the aggregate – to see if the new sample is still far enough into the tail. It rejects new samples until it finds a sample that satisfies the constraint.

4.2 Motivation

In MCDB-R, each database sample is called a DB instance. MCDB-R borrows tuple bundle idea from MCDB to perform efficient query processing on these large number of DB instances. A tuple bundle stores all the samples (DB instances) for a given dimension of the data. Each tuple bundle is uniquely identified by its seed identifier (Section 2.4). Because of tuple bundles, the query is run only once, instead of once for
each DB instance. Thus, bundling improves the performance significantly compared to independent execution of the query on each of the DB instances. The sample sizes used in MCDB-R are larger than those in MCDB because the Rejection algorithm consumes significantly more samples. The samples in a tuple bundle are generally exhausted when the Rejection algorithm is replacing a sample with low probability. It can happen that the aggregate value on the data is very close to the constraint. Then the Rejection algorithm should find another sample very close to the sample value it is trying to replace. Otherwise the constraint will not be satisfied. Since the original sample is of low probability, the Rejection algorithm might not find a replacement until it rejects a large number of samples. Whenever a tuple bundle is out of samples the query is rerun to generate more samples. At each rerun the number of samples generated, \( num_s \), is doubled (unless a single tuple bundle size exceeds the page size, in which case \( num_s \) remains same). Since \( num_s \) is same for all the tuple bundles in a query run, each tuple bundle now stores double the samples and is approximately double in size. But most of the other tuple bundles may not require the new samples that are generated.

**Example:** When executing query Q2 (see Section 4.7), in second rerun the average number of samples used for each tuple bundle (or seed identifier) at Rejection algorithm is 150, where as the \( num_s \) is 4000. Most of the seed identifiers only need 150 samples, but due to the outliers that need 4000 samples, all seed identifiers have to store the 4000 samples. The framework of MCDB-R does not allow different sample sizes for different tuples. Even if the framework allows such option, it is not easy to predict beforehand which of the seed identifiers require more samples than others. This wastage of samples only increases with the number of reruns. At fourth rerun the number of used samples on average is 210 out of 16,000 samples generated. Though Normal VG function is not very expensive, generating so many unused samples had significant impact on the overall query execution time.
Figure 4-2. Modified query plan

Even if the sample generation process is inexpensive, the increase in data size will have an adverse effect on the response time. Figure 4-1 shows the simplified physical plan for query Q2. The samples are generated at the Instantiate operator (Section 2.5), and are passed through the plan to Tail sampler (Section 2.7) operator. The Tail sampler goes through the seed identifiers one by one in sorted order. The order is maintained by the Gibbs queue (Section 2.7.1). In most cases total tuples in Gibbs queue would not fit in main memory, therefore it writes them to disk. The solid arrow from Tail sampler to Gibbs queue signifies a new perturbation step (or Gibbs iteration) and reset of the Gibbs queue. The dotted arrow shows a query rerun, and is triggered when one of the seed identifiers runs out of samples. The query, when run on a 200 MB dataset with 185,000 tuple bundles and with 0.999 quantile as target tail area, writes approximately 25 gigabytes (GB) of data to disk in fourth rerun, when \( num_s \) is 16,000. For five Gibbs
iterations, the query writes and reads more than 1 TB to disk. See Section 5 for more
statistics on the execution of this query.

4.3 Solution Outline

The technique we present in this chapter allows the system to serialize the sample
generation process and delay the actual sample generation as long as possible.
The new query execution avoids the movement of the large tuple bundles through
some expensive operations like sorts and joins. The query plan in Figure 4-1 can
now be modified to that in Figure 4-2. In the new plan Instantiate(S) denotes
Instantiate-serialize. In this modified operator, instead of generating the samples
as before at regular Instantiate, the VG function (Section 2.3) parameters are serialized
into a string, called SVGF for Serialized VG Function, and kept along with the Seed
attribute (Section 2.4). Instantiate(G) de-serializes the SVGF string and generates the
requested number of samples. It can be seen in Figure 4-2 that the sample generation
is done after the Gibbs queue. Thus, instead of large number of samples now only the
string SVGF is written to the disk. SVGF string is typically less than 100 bytes in size.
For queries that does not have a join on stochastic attributes, and hence are similar in
structure to query Q2, this serialization eliminates the requirement of query reruns. Now
we can generate variable number of samples for each seed identifier separately, which
eliminates the generation of large number of unused samples for many seed identifiers
when only very few seed identifiers require them.

4.4 Serialization

In this section, we discuss the process of serializing the VG function and the
parameters for a given seed identifier. We refer to this representation as SVGF as noted
previously. For serializing the VG function we first store the name of the function and
then all the parameters that are supplied to the function. The schema of the parameters
needs to be stored as well. A VG function framework in MCDB-R is very generic. A VG
function can take multiple tuples from parameter (inner) tables and one tuple from outer
Figure 4-3. A VG function process

Figure 4-4. Serialization of VG function
table for generating a single sample. Instantiate picks the required attributes from these
tuples and creates a parameter array, that is then passed to the VG function. Figure 4-3
shows the VG function process within the Instantiate. Recreating this sample generation
process requires exact simulation of passing the parameters, as done in the Instantiate.
Serialization format is shown in Figure 4-4. After storing the VG function name we store
the schema of the parameter array. Then we store the number of times the parameter
array is passed, and then the parameter arrays themselves. The SVGF is created during
the first execution of the Instantiate. During this first execution, the VG function is called
as before except that only one sample (which is discarded later) is generated and SVGF
is formed. Once SVGF is created, it is stored as a string in the tuple as part of the Seed attribute related to the VG function. A single VG function call can produce a sequence of values as part of the sample, resulting in multiple stochastic attributes. Therefore the relationship between a stochastic attribute and its location in the sequence of values is stored in the stochastic attribute. Since a single tuple can have multiple Seed attributes, each stochastic attribute also stores the particular Seed from which it is generated.

<table>
<thead>
<tr>
<th>SEED</th>
<th>Price</th>
<th>New Price</th>
<th>Qty</th>
<th>α</th>
<th>α’</th>
<th>β</th>
<th>β’</th>
</tr>
</thead>
</table>

Figure 4-5. Bayesian VG: Input serialization

| SEED | NULL | NULL | NULL | α | α’ | β | β’ |

Figure 4-6. Bayesian VG: Inner input

| SEED | Price | New Price | Qty | NULL | NULL | NULL | NULL |

Figure 4-7. Bayesian VG: Outer input

Figures 4-5 through 4-8 show the serialization process for a Bayesian VG function (see query Q1 in Section 4.7). This VG function takes seven parameters, and of them four are passed through the inner input pipe and other three are passed from outer input.
Figure 4-8. Serialization of Bayesian VG input process

pipe. Figure 4-5 shows the schema of the parameter array. First attribute is the seed identifier, next three are the parameters from outer input. The last four parameters for the two Gamma functions in the Bayesian model. The parameters from inner and outer inputs are sent to VG function separately through different arrays, and different calls to the TakeParams(). Instantiate maps the inner and outer input tuples into the parameter array. Figures 4-6 and 4-7 show the transformation of an inner and an outer tuple into the array format. In the serialization step the parameter count is two, since we have one inner and one outer tuple. Figure 4-8 shows the SVGF created for the Bayesian VG function. First we store the name of VG function (Bayesian), then the schema of the parameter array and then the parameter count. In the parameter array, the unused fields are set to NULL. Before each value, a flag is stored to indicate whether that value exits or is it NULL in the array.

4.4.1 Incorporating Serialization in a Query Plan

Predicates on stochastic attributes are generally pushed down the query plan, and are evaluated just after the Instantiate. If the sample generation is serialized, then the samples are not generated at Instantiate. Therefore any predicates on the stochastic attributes should be moved to the operator at which the samples are generated. All the non-join predicates on stochastic attributes are passed to Tail sampler operator. These predicates are applied once the samples are generated inside Tail sampler.

4.4.2 Stochastic Join Predicates

Moving up the join predicate into Tail sampler in not recommended. With out a join predicate, a cross product is still performed. This could degrade the performance because a join predicate reduces the number of output tuples from a cross product. The join predicate can have a stochastic attribute in both sides of the join as in Figure 4-9
or just one side of the join as in Figure 4-11. In both query plans, the samples have to be generated before the Split operator (a MCDB specific operation that splits a stochastic tuple into multiple deterministic tuples, see Section 2.6.2), therefore we can only materialize all the operations just before it. If there are multiple Seed attributes in the tuple bundle before the Split operator, only the samples that are processed in Split
needs to be generated. A materialization just before Spilt operator is still useful as it avoids running the expensive Instantiate operation from the second run onwards.

As in the left subtree in Figure 4-9 if a stochastic attribute is not part of the join predicate, then there is no need to de-serialize that attribute until the Tail sampler. Given a query plan, we use the following rules to identify where in the query plan it is safe to materialize the operators. Rule 1: Any ancestor operator to a Split can not be materialized. Rule 2: After an Instantiate operator check if there are any joins. For any such joins, if the other side of the join predicate has a stochastic attribute, then materialize the operation just before the Split associated with the join. This case can be identified by traversing the other side of any joins in the path leading from Instantiate to Tail sampler in the query plan. Applying these rules result in modified query plans as shown in Figures 4-10 and 4-12. They are the modifications of query plans in Figures 4-9 and 4-11 respectively.

![Diagram](image)

Figure 4-11. Join on stochastic attributes on both sides

### 4.5 Efficient De-serialization

De-serializing the SVGF string is straightforward. First read the VG function name and create the function object. Then read the schema of the parameter array. After
that read the parameter arrays one by one and pass them to the VG function object. To make use of multiple cores in the machine, we can multi thread the sample generation process. This is easy since each sample has its own pseudo-random seed. This pseudo-random seed is computed through a hash function on the location of the sample in the sample stream and the PRNG seed from the Seed attribute. Each thread is given a range in the sample stream and the PRNG seed can generate the samples independent of other threads.

4.6 Related Work

This idea of serialization is closely related to compression in databases \([1, 29, 30, 36, 52, 74]\) which focuses on reducing the execution times by saving disk I/O. Since samples are of same type and are stored together in an array our setting is similar to storing columns in a column store. For integer or string data the bit-vector compression scheme \([1]\) can be used. Since the sample array is not sorted this seem to be the only method feasible. The serialization of VG function still offers more disk savings compared to bit-vector compression due to the large bit-vectors needed in MCDB-R. For example the size of SVGF for a Normal VG function is less than 100 Bytes, but a bit-vector for 16,000 values needs 16,000 bits is nearly 2 KB in size, a factor of 20. Compressing
floating point data is not very useful in MCDB-R since our data consists of random samples and maximum compression for random floating point data generally does not exceed a factor of 4 [48] in the best case.

4.7 Experiments

In this section we evaluate the serialization of VG functions through experiments. We conduct various experiments to test the efficiency and scalability of the new MCDB-R with serialization. Our aim is to show that using serialization in MCDB-R system allows it to execute queries much faster. The specific questions we want to answer here are:

1. Does serialization improve the efficiency of the queries?
2. How are the disk usage and number of samples generated effected?
3. Is the system more scalable now?
4. Does increasing the number of threads improve the sample generation after serialization?
5. How does the performance of the new system vary with different VG function types (short tailed vs long tailed)?

Experimental Setup: We use the Transaction Processing Performance Council Benchmark H (TPC-H) data generator [17]. The workstation used for the experiments has 32 GB of RAM, 4 720 GB Dell hard disks and 8 processing cores of 3.2 GHz each distributed over 2 chips. The system runs a 64-bit Ubuntu operating system, version 11.04. We added the serialization feature to the MCDB-R prototype implementation. It is written in C++ and consists of 30,000 lines of code. We use three queries for benchmarking. Query Q1 is based on query Q4 from [37], and query Q2 is from [6]. See Appendix 7 for SQL statements for both Q1 and Q2. Query Basic has the bare bones structure to use a VG function.

Query Q1. This query simulates the effects of a 5% increase in price on the profits. The change in customer demand due to new prices is estimated by using a Bayesian
VG function. A same prior distribution for demand is used for all customers. Bayesian methods are used to find the posterior distribution based on the parameters specific to the customer. The VG function outputs the possible customer demand for new price. We chose this query for benchmarking because it has an expensive VG function.

**Query Q2.** This is mainly used for testing correctness. The orders relation is made stochastic by adding normally distributed stochastic attribute. In the parameter table for the normal distribution, all mean and variance values are set to 1. The query is such that the result distribution is also normally distributed, and has a closed form solution.

**Query Basic:** This query has a single Instantiate before Tail sampler. The aggregate is on the sample value from the VG function. The Instantiate takes one outer table, and one parameter table. For this query, parameters are same for each of the tuples in the outer table. This query is used to compare different VG functions. The probability density functions used here range from those with no tail (Uniform), to long tailed distributions like Gamma and $\chi^2$.

Given this setup, there are six different variables we need to know for each test, in order to understand the results:

1. **query:** The query used in the test.
2. **DBsize:** The on disk size of the TPC-H dataset, in binary representation.
3. **numSeedIds:** The number of seed identifiers processed during the query execution.
4. **num\_db:** Number of DB instances for the query.
5. **gibbsIters:** Number of iterations at Gibbs looper.
6. **eliteDBPercent:** Number of DB instances retained after the end of each Gibbs iteration. The deleted DB instances are cloned from the retained elite instances.

We run five different tests. First test is executed on MCDB-R, with and without serialization. Rest of the tests are on MCDB-R with serialization. Second and third tests are useful in understanding the scalability of the new system. Test four helps us understand if the time taken for sample generation after serialization can be improved by
use more threads. Fifth test is useful in understanding the performance of serialization on long tailed distributions when compared to short tailed ones. Here is a concise description of the parameter values used for each test:

1. **Performance with and without serialization:** We use queries Q1 and Q2. DBsize is 200 MB. numSeedIds for Q1 is 93,000, and for Q2 is 46,000. \( num_{db} \) is 100. gibbsIters and eliteDBPercent are 5 and 50% respectively.

2. **Scalability with increasing DBsize:** We use two datasets with DBsize 200 MB and 20 GB. The value of numSeedIds for Q1 is 93,000, and for Q2 it is 46,000 on 200 MB dataset. The value is 9,100,000 for Q1, and is 4,600,000 for Q2 on 20 GB dataset. Rest of the parameters are same as in Test 1.

3. **Scalability with increasing \( num_{db} \):** The variable gibbsIters is varied from 10 to 400. Rest of the parameters are same as in Test 1.

4. **Parallel sample generation after serialization:** All the parameters are same as in Test 1. We include a multi-threaded implementation of sample generation at Instantiate(C). The number of threads is varied from 1 to 4.

5. **Performance for different VG functions:** The queries used here are query Basic with Uniform, Normal, Gamma, and \( \chi^2 \) VG functions. The value of DBsize is 20 MB, numSeedIds is 10,000, and \( num_{db} \) is 100. Values of gibbsIters and eliteDBPercent are 10 and 50% respectively.

**Results:** The results for these five tests are shown from Tables 4-1 to 4-6. We ran each test multiple times and listed the average values. Tables 4-1 and 4-2 show the data for Test 1. The performance improvement, when comparing MCDB-R with and without serialization is given in Table 4-1, under column Improvement Factor. Disk usage, and the number of samples generated/used for each execution are given in Table 4-2. The total disk usage values shown here are computed by keeping track of total page reads and writes during the execution of the query. Peak disk usage is calculated from the data input into the Gibbs queue. Wall clock times (in minutes) for Test 2 are given in Table 4-3. In Table 4-4, we have the wall clock times for Test 3. The times given are in seconds. Results for Test 4 are given in Table 4-5. Again, the times are given in seconds. Finally, Table 4-6 gives the execution times (in minutes) for Test 5.
Table 4-1. Wall clock execution times

<table>
<thead>
<tr>
<th></th>
<th>With Serialization</th>
<th>With out Serialization</th>
<th>Improvement Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>4.2 mins</td>
<td>24 mins</td>
<td>5.7</td>
</tr>
<tr>
<td>Q2</td>
<td>2 mins</td>
<td>32 mins</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 4-2. Disk I/O and sample statistics

<table>
<thead>
<tr>
<th></th>
<th>Total Disk I/O (in GB)</th>
<th>Peak Disk Usage (in GB)</th>
<th>Total Samples (in Billions)</th>
<th>Avg. Gen. Per Seed Id</th>
<th>Avg. Used Per Seed Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 with Serialization</td>
<td>9</td>
<td>1</td>
<td>0.12</td>
<td>1,300</td>
<td>500</td>
</tr>
<tr>
<td>Q1 without Serialization</td>
<td>74</td>
<td>4</td>
<td>0.65</td>
<td>7,000</td>
<td>500</td>
</tr>
<tr>
<td>Q2 with Serialization</td>
<td>10</td>
<td>1</td>
<td>0.061</td>
<td>1,326</td>
<td>510</td>
</tr>
<tr>
<td>Q2 without Serialization</td>
<td>1374</td>
<td>55</td>
<td>3.4</td>
<td>74,000</td>
<td>510</td>
</tr>
</tbody>
</table>

The parameter values used for different VG functions are given in first row. For example, the Normal VG function has mean \( \mu \) set to 1, and variance \( \sigma^2 \) set to 1.

**Discussion:** Test 1: Table 4-1 shows a significant improvement when serialization is added. For Q2, the performance improvement is more pronounced because the normal VG function generated rare samples more frequently than the Bayesian VG function. In Table 4-2 we can see the evidence for this. When using serialization, query Q1 generates one fifth of samples compared to query Q2 without using serialization. For query Q2 the ratio of samples generated is 50. Execution of query Q2 without serialization had to rerun the plan three times for generating extra samples (not given in the table). This is significant because every new run results in two extra passes over

Table 4-3. Scaling with database size

<table>
<thead>
<tr>
<th>Size</th>
<th>200 MB</th>
<th>20 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>5.5 mins</td>
<td>428 mins</td>
</tr>
<tr>
<td>Q2</td>
<td>2 mins</td>
<td>241 mins</td>
</tr>
</tbody>
</table>
the samples. More reruns also means more unprocessed samples. Even when only one seed identifier requires extra samples they are generated for all other seed identifiers as well.

Tests 2 and 3: Table 4-3 shows that the system is scalable and can be run on large databases with millions of tuples. For both queries a factor of 100 in dataset size seem to result in approximately a factor of 100 in the execution time. The execution time seems to scale linearly. In Table 4-4 we can see that for both queries Q1 and Q2, the response time increases linearly as we vary the number of DB instances. Test 4: The performance is worse for the multi-threaded implementation. A major reason for this could be using WELL512 (Well Equidistributed Long-period Linear, 512 bytes) [46] random generator. WELL512 is initialized once for each thread, and this initialization is expensive as it buffers 512 random bits. The query is run for 100 DB instances, so in most cases we generate only 200 samples at a time. If we use 4 threads, each thread now produces 50 samples and so may not use a significant portion of those 512 bits. Using a different random generator could avoid this problem. Test 5: The execution time in Table 4-6 shows that, as expected, the query with Uniform VG function finishes faster than others with longer tails. The query with Gamma VG function, with low shape

Table 4-4. Scaling with DB instances (time in seconds)

<table>
<thead>
<tr>
<th>num_db</th>
<th>10</th>
<th>100</th>
<th>200</th>
<th>400</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>47</td>
<td>252</td>
<td>476</td>
<td>958</td>
</tr>
<tr>
<td>Q2</td>
<td>40</td>
<td>120</td>
<td>206</td>
<td>376</td>
</tr>
</tbody>
</table>

Table 4-5. Parallel sample generation

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>252</td>
<td>332</td>
<td>373</td>
</tr>
<tr>
<td>Q2</td>
<td>120</td>
<td>150</td>
<td>148</td>
</tr>
</tbody>
</table>

Test 4: The performance is worse for the multi-threaded implementation. A major reason for this could be using WELL512 (Well Equidistributed Long-period Linear, 512 bytes) [46] random generator. WELL512 is initialized once for each thread, and this initialization is expensive as it buffers 512 random bits. The query is run for 100 DB instances, so in most cases we generate only 200 samples at a time. If we use 4 threads, each thread now produces 50 samples and so may not use a significant portion of those 512 bits. Using a different random generator could avoid this problem. Test 5: The execution time in Table 4-6 shows that, as expected, the query with Uniform VG function finishes faster than others with longer tails. The query with Gamma VG function, with low shape

Table 4-6. Performance for different VG functions (time in minutes)

<table>
<thead>
<tr>
<th>VG Function</th>
<th>Uniform (a=0, b=0)</th>
<th>Normal (μ=1, σ^2=1)</th>
<th>Gamma (k=1, β=6)</th>
<th>χ^2 (k=6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>34</td>
<td>40</td>
<td>46</td>
<td>41</td>
</tr>
</tbody>
</table>
parameter and a long tail, takes the most time. From this test, we can see that MCDB-R with serialization works well for heavy tailed distributions also.

**Conclusion:** Four of the five experiments we ran gave expected results. We can conclude from Tests 1, 2, 3, and 5, that serializing VG function in MCDB-R is extremely useful in improving the query performance and scalability of the system. More work needs to be done on making the sample generation (after de-serialization) parallel and faster. Test 4 showed that our current multi-thread approach for this operation is not efficient.
CHAPTER 5
REPLACING AN EXTREME SAMPLE

5.1 Motivation

MCDB-R system uses sampling based Monte Carlo simulations to perform risk analysis on uncertain data. In this system the data uncertainty is modeled through parametric probability distribution functions. The whole database is seen as a high dimensional joint distribution. Risk analysis on this data can be done through observing the low probability regions – the upper and lower tails of the distribution function. Gibbs cloning method is used to sample from the tail of this high dimensional distribution.

Gibbs cloning has two steps – cloning and perturbation, as explained in Section 2.1. Perturbation step is achieved through Gibbs sampler, a Markov Chain Monte Carlo method. Gibbs sampler processes huge number of samples during a single query execution. Even for a moderate sized dataset the number of samples generated can be in billions, as shown in some experiments from previous chapter. Since we process such huge number of samples, the chances of the occurrence of an extreme sample (one which occurs with very low probability) in the sample stream is high. Such a sample is easily accepted at the Gibbs sampler as a good replacement if it is in the correct side of the tail. Being on the same side of the target tail the sample will pass the constraint easily. But replacing that sample later, during the next Gibbs sampler iteration, might require another sample of nearly equal value (and hence very low probability of occurrence). For example, if the probability of occurrence of a sample we are replacing is $10^{-6}$, then the rejection algorithm (Section 2.1.2) will probably reject a million samples before finding a feasible replacement for it. Replacing one such extreme sample could exhaust the sample array from the current query run. We may even require numerous query reruns. Remember that during these reruns sample generation is done for rest of the tuples as well, not just the tuple that ran out of samples. Therefore, replacement process will make the query response time extremely high. In this chapter we will look
Table 5-1. Poisson random relation

<table>
<thead>
<tr>
<th>Id</th>
<th>Value</th>
<th>Sample stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>7</td>
<td>5, 7, 8,...</td>
</tr>
<tr>
<td>b</td>
<td>8</td>
<td>12, 7, 8,...</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td>4, 3, 4,...</td>
</tr>
<tr>
<td>d</td>
<td>9</td>
<td>8, 9, 7,...</td>
</tr>
<tr>
<td>e</td>
<td>11</td>
<td>9, 10, 8,...</td>
</tr>
</tbody>
</table>

Table 5-2. After a phase of rejection sampler

<table>
<thead>
<tr>
<th>Id</th>
<th>Value</th>
<th>Sample stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>7</td>
<td>8, 5, 6,...</td>
</tr>
<tr>
<td>b</td>
<td>12</td>
<td>7, 8, 6,...</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>3, 4, 5,...</td>
</tr>
<tr>
<td>d</td>
<td>8</td>
<td>9, 7, 7,...</td>
</tr>
<tr>
<td>e</td>
<td>9</td>
<td>10, 8, 9,...</td>
</tr>
</tbody>
</table>

at an effective solution for this extreme value replacement problem in order to lower that response time.

**Example:** Consider the random relation Poisson in Table 5-1. It has the stochastic attribute Value and a Poisson VG function (Section 2.3) associated with it. Only one database instance is listed for the stochastic attribute to simplify the example. Lets say we want to perturb this database instance using the sample stream available. We need to replace the Value attribute in each tuple. The aggregate here is a summation and let the constraint value be 23. That is, \( \text{sum(Value)} \) must be greater than or equal to 23. The instance of the relation after perturbation is shown in Table 5-2. The \( \text{sum(Value)} \) is still 23, and the tuple with id 'b' has the sample value 12. The accepted sample values for tuples following tuple 2 are lower than their existing values because the lower values still satisfy the constraint on the aggregate. For tuple 'c' the value 5 is replaced by 4 since the new \( \text{sum(Value)} \) 25 is still larger than 23, the constraint. Similarly for tuple 'd', 9 is replaced by 8, and for tuple 'e' 11 is replaced by 9. Now in the next perturbation phase, the rejection algorithm will require a value of 12 or more for tuple 'b' as anything lower will make the aggregate \( \text{sum(Value)} \) go lower than 23. Finding such a sample is difficult because of its low probability. If we assume that we need 30,000 samples to
find another value greater than or equal to 12, then in the current MCDB-R we need to generate same number of samples for other tuples as well. The performance can be improved drastically if we can eliminate producing such unnecessary samples.

Fortunately, in queries without a join on stochastic attributes, the serialized VG function is accessible at the rejection algorithm. Hence, processing very large number of samples at the rejection algorithm does not require a query rerun, as it can generate samples on the fly whenever necessary. But this replacement process can be quite expensive in other queries as it might require large number of query reruns. Note that page size of the system will limit the maximum number of samples in a single tuple bundle and in turn in a single query run. A page size of 1 MB can at most hold 130,000 samples. To look at a million samples we need 8 runs even with a page size of 1 MB. It is possible that we might encounter multiple such extreme samples during the execution of a single query, each requiring tens of query reruns. If the number of tuples in the stochastic relation is large then there is a good chance of encountering multiple extreme value situations in a single perturbation step. Without a proper method to speed up this process the queries will be very expensive.

5.2 Solution Outline

In this section we give the outline of an approach to solve this extreme value problem efficiently. The first step is to eliminate the sample generation for unused tuples in each query rerun. Since we need extra samples only for a single seed identifier (Section 2.4), unless there is a self-join, we can eliminate tuples with rest of the seed identifiers from the Instantiate (Section 2.5). For other seed identifiers which are not eliminated, we need to generate only those samples that are currently in use by a DB instance. For the given seed identifier we might need to produce tens of millions of samples. Since all the samples can not fit in a single page we need to devise a get around. One possibility is to perform a Split operation as soon as the samples are generated. Though this eliminates the problem of overflow, we still need to carry a
very long bit-string. A bit-string for 10 million samples is just above 1 MB, but is still manageable. Even this solution becomes unmanageable if we need larger number of samples.

A more clean solution is to break the large tuple bundle into numerous smaller ones just after the Instantiate. Once these smaller tuples are passed through the query plan, they can be stitched back inside the Tail sampler (Section 2.7). We will keep sequence identifiers to remember the order in which the smaller tuples need to be grouped together. This strategy will not affect the correctness of the result except when the operator is an aggregate or a self-join. We can safely assume that there are no aggregates before Tail sampler as MCDB-R does not accept such queries. In case of a self-join, two tuples from the same seed identifier should be joined only if their sequential identifiers match. This constraint ensures that the join does not produce spurious tuples.

5.3 Filtering

Filtering out the tuples of unnecessary seed identifiers is very effective in reducing the resources required for finding an extreme sample. Filtering will reduce the CPU time since we avoid producing samples for the filtered out tuples. The movement of large number of tuples in the query plan will also reduce the memory and disk usage. This process is illustrated for a simple query plan in Figures 5-1 and 5-2. Figure 5-1 shows the query plan for the normal execution. This query has a single VG function serialized at the right leaf node, the Materialized Instantiate. After Instantiate, the tuples are Split before being sent to the join. Figure 5-2 provides the new query plan with a selection above the Instantiate. This selection will allow only tuples that have the required seed identifier to go to the next operator. The necessary seed identifier is identified at the Tail sampler and then is sent to the Instantiate for the selection process.

Note that the filtering can only be applied at the particular Instantiate that generated the seed identifier corresponding to the extreme sample. If multiple stochastic attributes
exist in the query plan we have multiple Instantiates, one for each stochastic attribute. All other Instantiates except the one that produced the extreme sample can not filter out any of their tuples. Fortunately, for any such stochastic attributes we only need the active samples; only $num_{db}$ samples for each tuple. Moreover only tuples that join with the seed identifier of the extreme sample remain by the time tuples reach the Tail sampler. This is because if the other stochastic attributes are in the same relation as the extreme sample, then the filtering on the seed identifier of the extreme sample will also
eliminate many other seed identifiers from the other stochastic attribute. The second possibility is that the other stochastic attribute exists in a different relation than the one with extreme sample. In such cases the relation with extreme sample, and the other stochastic relation are joined. Since the relation with extreme sample will have very few tuples due to filtering, the join result itself will be smaller accordingly. Hence, existence of other stochastic attributes will not degrade the performance significantly.

_Self-joins_ need to be handled separately, as the filtering can not happen until after the join in the query plan. If a query plan has a self-join on the stochastic attribute that contains the extreme sample, we can not filter out the tuples until after the self-join. It can also happen that the tuples which have the extreme sample can have multiple seed identifiers from the same relation even without a self-join. This can happen due to a self-join on attributes other than the stochastic attribute. If these seed identifiers belong to the same relation as the one that generated the extreme sample, then we need to follow the same process – filtering is delayed until after the self-join.

### 5.4 Fragmenting a Large Tuple Bundle

Since a VG Function can generate multiple tuples as part of the same sample we maintain a VG identifier \( \text{vg-id} \) to keep track of that order. We need a third identifier to differentiate between multiple tuples with the same seed if they occur before Instantiate.

**Figure 5-3. Splitting a large tuple bundle**

Since a VG Function can generate multiple tuples as part of the same sample we maintain a VG identifier \( \text{vg-id} \) to keep track of that order. We need a third identifier to differentiated between multiple tuples with the same seed if they occur before Instantiate.
This scenario can happen when there is a join before the Instantiate. We call this identifier a tuple identifier $t$-$id$. Figure 5-3 shows an example for fragmenting a very large tuple bundle into smaller pieces. In this example we keep the number of samples in the smaller tuple bundles at 10,000. Instead of a single tuple with 10 million samples, we will have 100,000 tuples with 10,000 samples each. The sequence identifier $seq$-$id$ is from 1 to 100,000. The $t$-$id$ is same for all the smaller tuples as we only have one original tuple before Instantiate. The $vg$-$id$ is same here since a single input tuple to Instantiate produces a single output tuple. Even a million samples might not be enough for some extreme values. Therefore, we repeat the process with ten times more samples than the previous run. If a satisfactory sample is still not found, we will increase the sample array size again by ten times. We repeat this until we find that replacement.

5.5 Implementation Details

![Query plan with Sample router: normal mode](image)

Incorporating fragmentation in the existing query plan is non trivial. After an extreme sample is found, any changes done to the query plan should be reset. Filtering is fairly easy to add to the existing query plan since it only requires a select operator on the seed identifiers. The selection predicate is sent by the Tail sampler to the Instantiate.
When the extreme sample is found after executing the new query plan, the Instantiate is again informed of the finding and the selection predicate is dropped.

Unlike filtering, fragmentation requires some major changes in the query plan. In our implementation, we create a new flow for the tuples in the Tail sampler. Instead of sending the tuples to Gibbs queue, we will pass them to another data structure, called a Sample router. This data structure will sort the incoming tuples according to the seq-id’s and passes the Gibbs sampler only the tuples with same seq-id. If necessary, Gibbs sampler will request the next set of tuples with next seq-id. In normal execution mode the Sample router just acts as a relay and does nothing. See Figure 5-4 for an example query plan, with a Sample router. The compiler, when it first sees a stochastic join in the query plan, will modify the plan to include the Sample router. This addition is done even before any execution is started. If an extreme sample does not occur, then the Sample router will always be in the normal execution mode. The Tail sampler will notify the Sample router whenever the execution mode needs a change. The example query plan in active mode is shown in Figure 5-5.
5.5.1 TSSeed and Sample Generation

After the extreme sample is found, the query is restarted in the normal mode. All iteration numbers for seed identifiers are stored in TSSeed at the end of each run. During the new query run, $num_{mc}$ samples are generated at Instantiate. The starting iteration number for this sample generation is the highest iteration number assigned to a DB instance. That starting iteration number will be the low iter. number in next query run. Any unassigned iteration numbers between the low iter. number and max. used iter. are not used. This generally does not create a problem because their number is usually small compared to $num_{mc}$. Unfortunately, just after an extreme sample is found, this assumption does not hold true. More often than not, $num_{mc}$ is much smaller than the difference between low iter. number and max. used iter.
Figures 5-6 and 5-7 show the TSSeed (Section 2.6.1) object of a seed identifier with an extreme sample. The value at DB instance 2 in Figure 5-6 is the extreme sample, and its replacement is found only after processing more than 23,000 samples. In this example, num\textsubscript{mc} for the run after find the extreme sample will be 64. low iter.number is 18, max.used iter. is 23755, and the difference is 23737. One solution to this problem is to assign num\textsubscript{mc} a value at least double of the difference. This solution will only work until the difference between low iter.number and max.used iter. is larger than what a single page can hold. A better solution is to store the assigned samples on a persistent data structure, and generate only samples after max.used iter.. We can restore the assigned samples for that seed identifier from the data structure. This approach might get expensive if there are large number of extreme samples in a single Gibbs sampler loop. Then, the data structure needs to store the samples of all seed identifiers with a extreme sample. A simpler approach is to change how the samples are generated at Instantiate using TSSeed. The TSSeed creates the PRNG seed array used in Instantiate from iteration numbers and Seed Id. The range of iteration numbers used is from low iter.number to num\textsubscript{mc}+low iter.number. This PRNG seed array determines what sample values come out of the VG function. We change the iteration numbers used. First num\textsubscript{db} iteration numbers are taken as is from the assigned iteration numbers. Rest of the num\textsubscript{mc} – num\textsubscript{db} iteration numbers start from max.used iter.. Implementation of this method only requires changes to TSSeed object, GetSeeds function. No other changes are required in the code.

5.5.2 How to Identify an Extreme Sample?

During execution, we need to identify a sample as extreme in order to invoke our specific method. One way to identify an extreme sample is to do some statistical analysis, and find out the probability of occurrence of the replacement sample. For that we first have to find the minimum (or maximum) sample value we need for a replacement. Once we have that, using the parameters of the VG function we can, in
theory, calculate the probability of finding a sample greater (or smaller) than that value. For a generic aggregate function, finding that minimum required value itself is non trivial. Instead, we can say that we need a sample with same value as the one we are replacing. This method of identification might result in many false positives since the actual value required might be easier to find. The special execution we use to find an extreme sample is very efficient compared to rerunning the whole query. But, too many false positives in this method will eliminate that efficiency advantage.

Instead, we can use a method based on run time statistics to identify an extreme sample. A sample can be designated as extreme if we can not find a replacement by the end of the sample array. But this again will create large number of false positives since it is possible to reach the end of sample array even for normal samples after multiple Gibbs iterations. To avoid that drawback we can say a sample is extreme when, even after a second rerun we do not find the replacement. We spent one entire rerun just to find that replacement. The other method is to use the number of samples consumed to find the replacement when we reach the end of the sample array. If at least 10% of the samples in the array are consumed without finding a replacement for a sample, then we can designate it as an extreme sample. In general, a decently large number like 10,000 might also suffice. But this can create problems during the first few reruns when the sample array size is still less than 10,000. A few reruns will be wasted until the number of samples seen to find the replacement exceeds 10,000. We think using a factor of sample array size like 10% is better than a fixed number like 10,000, since the fixed value might need to be recalculated for a different page size. We experimentally evaluate two different ways to identify an extreme sample. (1) A sample is called extreme if a replacement is not found at the end of two query runs. and (2) if a sample consumed more samples than 10% of the current sample array.
5.6 Experiments

In this section we evaluate our solution. Our aim is to compare the performance of our solution against the current approach as implemented in the MCDB-R prototype. We also want to show that our solution works well on large data sizes, and fast growing aggregate functions. Different aggregate functions will result in different number of extreme samples, therefore it is interesting to observe the performance by varying aggregate function. Finding an optimal page size here will directly affect the query response time: A very small page size might result in too many reruns due to small sample array size, where as a very big page size might result in throwing away too many samples from the array (whenever an extreme sample is encountered and the query is rerun). The number of extreme samples are expected to increase as we move further into the tail. This will show that our solution is extremely important. The specific questions we want to answer here are:

1. How much is the performance improvement due to our solution?
2. How scalable is the new system?
3. How does the new system perform on different aggregate functions (linear vs quadratic vs cubic)?
4. Does varying the system page size affect the performance?
5. What is the rate of increase in the number of extreme samples as we move further into the tail?

Experimental Setup: We use the TPC-H benchmark data generator [17]. The workstation used for the experiments has 32 GB of RAM, 4 720 GB Dell hard disks and 8 processing cores of 3.2 GHz each distributed over 2 chips. The system runs a 64-bit Ubuntu operating system, version 11.04. We implemented our solution in the prototype MCDB-R system. We use two queries for benchmarking. The query Q3 below is based on query Q10 from [38]. See Appendix for complete SQL statements for both the queries used. All queries are based on modified TPC-H schema.
Query Q3. In this query, total revenue generated by all customers in Japan is calculated. The schema is modified such that the customer information in the orders relation is not fully correct due to errors in the data integration process. A new relation is created to incorporate this error in the database. This relation has the actual customer key and a list of possible customer keys as a stochastic attribute. Each of the possible customer keys has a probability value. The number of possible customer keys is limited to 10, one of them being the correct key. The probability values are calculated according to the following formula \( \frac{1}{2^i} \), where \( i \) varies from 1 to 10. The resulting distribution here has finite domain. A discrete choice VG function is used to generate samples from the possible key and its probability from this new parameter relation.

In [38] the highest probability of 0.5 is always given to the correct key. When run with these values we will never encounter a situation where a sample replacement requires huge number of samples. Here we modified the table such that in 50% of the tuples the correct key is given the lowest probability of \( \frac{1}{2^{10}} \). This modification is done in order to create a situation where two samples with the correct customer key value can occur very far away from each other in the sample stream. Such a situation can result in a sample which is difficult to replace – in other words, an extreme sample.

Query Q4. Here we calculate the penalty due to shipping delays to the customers. We have a \textit{shiptime} relation which has for each customer the number of days within which the item should be delivered. The relation \textit{avg\_shiptime} has the average shipping time for a given item, and this is the parameter for the Poisson() VG function. The relation \textit{penalty} has the penalty added according to the number of days of delay in shipping. For these experiments, the penalty rate we use for the delay increases as the number of days of delay: \( \text{penalty}(i) = i \times j \) where \( i \) is the number of days of delay, and \( j \) is the penalty rate per day with \( j = \lfloor i/5 \rfloor \). In this function the rate of penalty increases at 5 day intervals.
Given this setup, there are six different variables we need to know for each test to understand the results:

1. query: The query used in the test.
2. numSeedIds: The number of seed identifiers processed during the query execution.
3. PageSize: Size of the system page. Limits the size of the sample array. This value is set to 128 KB for our experiments, unless specified otherwise.
4. num\textsubscript{db}: Number of DB instances for the query. This is set to 100 for all tests in this section.
5. gibbsIters: Number of iterations at Gibbs looper. This value is 5, unless specified otherwise.
6. eliteDBPercent: Number of DB instances retained at the end of each Gibbs iteration. The deleted DB instances are cloned from the retained elite instances. We have this value at 50% for all the tests.

We run the following five tests. In first test we compare the original MCDB-R (with serialization) with MCDB-R with our solution. We test the new system on a 20 GB dataset in Test 2, and different aggregate functions in Test 3. In Test 4, we vary the system page size. We calculate the number of extreme samples with changing Gibbs iterations in Test 5. We list how the variables are set for each of the tests:

1. Performance improvement due to our solution: Both queries Q3 and Q4 are used here. The values of numSeedIds is 1000, PageSize is 128 KB, num\textsubscript{db} is 100, gibbsIters is 10, and eliteDBPercent is 50%.
2. Scalability with increasing numSeedIds: We used query Q4. The value of numSeedIds is varied from 8000 to 40,000, and gibbsIters is set to 5. Rest of the variables are same as in Test 1.
3. Performance with different aggregate functions: We used query Q4. The value of numSeedIds is 20,000, and gibbsIters is set to 5. Rest of the variables are same as in Test 1.
4. Effect of PageSize on performance: We used query Q4. The value of numSeedIds is 10,000, and gibbsIters is set to 5. The value of PageSize is varied from 32 KB to 512 KB. Rest of the variables are same as in Test 1.
5. Number of extreme samples with increasing gibbsIters: We used query Q4. The value of numSeedIds is 8000, and gibbsIters is varied from 2 to 10. Rest of the variables are same as in Test 1.

Results: The results for these five tests are shown in Tables 5-3 to 5-11. We ran each test multiple times and listed the average values. Results of Test 1 are given in Tables 5-3, and 5-4. Table 5-3 shows the performance of original MCDB-R and our solution with two definitions (Section 5.5.2) of extreme samples. Type-1 is when a replacement is not found at the end of two query runs. Type-2 is when a sample consumed more samples than 10% of the current sample array. For the same test, the number of samples identified as extreme are shown in table 5-4. Test 2 results are given in Tables 5-5, 5-6 and 5-7. Table 5-5 compares the performance of original MCDB-R with Type-2 variant of our solution. We do not use Type-1 in this and later tests since we found that Type-2 has a better performance. The number of total samples generated in this test are given in Table 5-6. Table 5-7 gives the number of query reruns required for each of the query executions listed in Table 5-5. The values give both the number of full query reruns, and the number of reruns with filtering (i.e. the number of extreme samples). In each cell, the number of the reruns used for replacing an extreme sample is given in brackets ( ). For example input size with 8000 random tuples we have 12 Type-2 extreme samples, and 10 query reruns.

Table 5-8 shows results for Test 3. The running times for the query executed with different aggregate functions, and the number of extreme samples identified for each execution are listed. Results of Test 4 are show in Tables 5-9 and 5-10. Table 5-9 shows the query response time with changing PageSize (and in turn maximum sample array size) in the system. Table 5-10 lists the number of total runs required to finish the query and the number of extreme samples in each run. Table 5-11 shows the results from Test 5.

Discussion: Test 1: As explained above, the query Q3 has a distribution with finite domain on its stochastic attribute. This limits the chances of occurrence of an
Table 5-3. Wall clock running times (in seconds)

<table>
<thead>
<tr>
<th>Query</th>
<th>Original</th>
<th>Type-1</th>
<th>Type-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q3</td>
<td>43</td>
<td>43</td>
<td>41</td>
</tr>
<tr>
<td>Q4</td>
<td>219</td>
<td>135</td>
<td>75</td>
</tr>
</tbody>
</table>

Table 5-4. Number of extreme samples identified

<table>
<thead>
<tr>
<th>Query</th>
<th>Type-1</th>
<th>Type-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Q4</td>
<td>7</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5-5. Performance with increasing input size (in minutes)

<table>
<thead>
<tr>
<th>Size</th>
<th>8000</th>
<th>20,000</th>
<th>40,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>15</td>
<td>102</td>
<td>246</td>
</tr>
<tr>
<td>Type-2</td>
<td>7.3</td>
<td>28</td>
<td>110</td>
</tr>
</tbody>
</table>

Table 5-6. Samples generated (in Billions)

<table>
<thead>
<tr>
<th>Size</th>
<th>8000</th>
<th>20,000</th>
<th>40,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>1.5</td>
<td>14.9</td>
<td>21.7</td>
</tr>
<tr>
<td>Type-2</td>
<td>0.63</td>
<td>3.2</td>
<td>10.7</td>
</tr>
</tbody>
</table>

Table 5-7. Query reruns and extreme samples

<table>
<thead>
<tr>
<th>Size</th>
<th>8000</th>
<th>20,000</th>
<th>40,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>25(0)</td>
<td>94(0)</td>
<td>69(0)</td>
</tr>
<tr>
<td>Type-2</td>
<td>10(12)</td>
<td>21(24)</td>
<td>34(47)</td>
</tr>
</tbody>
</table>

Table 5-8. Performance with change in aggregate attribute

<table>
<thead>
<tr>
<th>Aggregate attribute</th>
<th>penalty</th>
<th>penalty*5</th>
<th>penalty^2</th>
<th>penalty^3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (minutes)</td>
<td>20.3</td>
<td>26.1</td>
<td>32.8</td>
<td>89.1</td>
</tr>
<tr>
<td>Num. Extreme Samples</td>
<td>15</td>
<td>20</td>
<td>24</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 5-9. Performance with changing max. samples (in minutes)

<table>
<thead>
<tr>
<th>Max Samples</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
<th>16,000</th>
<th>32,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min Page Size</td>
<td>32KB</td>
<td>64KB</td>
<td>128KB</td>
<td>256KB</td>
<td>512KB</td>
</tr>
<tr>
<td>Time</td>
<td>32.2</td>
<td>26.3</td>
<td>23.1</td>
<td>31.4</td>
<td>27.6</td>
</tr>
</tbody>
</table>

Table 5-10. Execution parameters with changing max. samples

<table>
<thead>
<tr>
<th>Max Samples</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
<th>16,000</th>
<th>32,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num. Runs</td>
<td>30</td>
<td>25</td>
<td>31</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Extreme Samples</td>
<td>24</td>
<td>20</td>
<td>16</td>
<td>22</td>
<td>20</td>
</tr>
</tbody>
</table>
Table 5-11. Number of extreme samples with increasing Gibbs iterations

<table>
<thead>
<tr>
<th>Iterations</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-2</td>
<td>3</td>
<td>14</td>
<td>24</td>
<td>45</td>
<td>72</td>
</tr>
</tbody>
</table>

Table 5-4 shows that for query Q3, we only have 1 extreme sample in Type-2, and none in Type-1. We can clearly see that there is almost no improvement on its execution time in new system over the old system. Unlike query Q3, query Q4 processes samples from a Poisson distribution, which has a significant tail. The number of extreme samples for Q4 are 7 and 13 for Type-1 and Type-2, respectively. We can see the execution time improvements for this query on both types. Type-2 identification clearly outperforms Type-1 since we do not waste an extra query run to identify an extreme sample. In Type-2, we may identify some samples as extreme even if they are not that extreme. This does not affect performance much since our special execution module to replace extreme samples is quite fast. For query Q4 the execution time difference between our special module and a full query run is approximately two orders of magnitude (1:100 ratio). We can see that the execution time improvements from Table 5-3 are proportional to the number of extreme samples identified, as given in Table 5-4. Therefore, we can infer that Type-2 performs better than Type-1 because of faster identification of extreme samples, and avoiding the extra query run it takes Type-1 to identify a sample as extreme. From now on, all the experiments use only Type-2 identification of extreme samples.

**Test 2:** The execution time in Table 5-5 shows that the performance of the new system is better than the old one consistently over varying input sizes. As expected, the numbers shown (of samples generated) in Table 5-5 are proportional to the corresponding execution times shown in Table 5-6. A stronger correlation can be seen between the number of query runs in Table 5-7 and the execution time in Table 5-5. This is not surprising since our module to process extreme samples takes very less time when compared to a total query run.
Test 3: From Table 5-8 we see that as expected the number of extreme samples is high for fast growing functions. The performance of the system is decent even for cubic function. Test 4: The execution times show that the performance is best with the maximum sample array size at 8000. Test 5: For first two Gibbs iterations, the number of extreme samples is 3, and for first four iterations it is 14. This value 14 is inclusive of first two iterations as well, i.e. the number of extreme samples in third and fourth iterations is $14 - 3 = 11$. As expected, the number of extreme samples increases as we move towards the tail.

Conclusion: We conclude that Type-2 variant of our solution is the best. The performance improvement is significant in queries with long tailed VG functions, and the solution works well on large datasets, and fast growing aggregate functions. As we can see from Test 5, the number of extreme samples increases as we move further towards the tail. Therefore, adding our special module to existing system is very essential.
CHAPTER 6  
ANTI JOIN

6.1 Motivation

Anti-Join operator is part of the standard SQL. It is required to execute SQL clauses 
\texttt{NOT IN} and \texttt{NOT EXISTS} in sub-queries. An anti-join of a left relation \( L \) and right relation \( R \)  
\((L \triangleleft R)\) returns tuples in \( L \) which do not match any tuples in \( R \) on the predicate. Consider the tables Orders (ORD ID, CUST ID, AMOUNT) and Customers (CUST ID, CUST NAME) in Figure 6-1. Suppose we execute the anti-join query, Orders \( \triangleleft \) Customers, with the predicate on Orders.CUST ID and Customers.CUST ID. The final output of the Orders \( \triangleleft \) Customers is given in the table Output. The first tuple in Orders with CUST ID value 21 appears in Output since there is no tuple in Customers with CUST ID as 21. Second tuple in Orders with CUST ID value 23 does not appear in Output since it matches with the second tuple in Customers. This process of finding a match is repeated for each tuple in Orders.

\begin{tabular}{|c|c|c|}
\hline
ORD ID & CUST ID & AMOUNT \\
\hline
21 & 21 & 12 \\
22 & 23 & 22 \\
33 & 24 & 21 \\
44 & 54 & 25 \\
55 & 43 & 42 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
CUST ID & CUST NAME \\
\hline
27 & Alice \\
23 & Bob \\
24 & Carol \\
64 & Carlos \\
43 & Charlie \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|}
\hline
ORD ID & CUST ID & AMOUNT \\
\hline
21 & 21 & 12 \\
44 & 54 & 25 \\
\hline
\end{tabular}

Figure 6-1. Example: Anti-Join in a relational database

In a relational database an anti-join operation is straightforward to implement.  
Anti-join \( L \triangleleft R \) can be executed similar to a sort-merge join with some changes in the 
merge step. Sort both relations on the attributes in the predicate. During the merge, 
if there is no match for a tuple in \( L \) with any tuple in \( R \), then output the tuple from \( L \). In 
MCDB-R, executing an anti-join on stochastic attributes is not as simple. An execution 
similar to a normal join, as explained in Section 2.6.2, is not enough. In the join, after a
Split operation on the stochastic attributes and a sort operation, a merge is performed on the sorted left and right relations. The join predicate is evaluated during the Gibbs sampling at Tail sampler (Section 2.7) operator. Evaluating a join predicate at Tail sampler requires only one pair of matching tuples, one each from right and left relations. An anti-join predicate on the other hand requires a tuple from left relation and all the matches from right relation.

A stochastic attribute in left relation of anti-join does not cause any difficulties because even when the sample is changed, the information in the tuple will be enough to execute the anti-join predicate. This is so only if the right relation does not have a stochastic attribute appearing in the predicate. If a stochastic attribute appears in the right relation the execution of anti-join is not as simple. When the current sample $v$ is replaced in a right tuple the anti-join predicate needs to know if the new sample is the first of its kind or are there any existing tuples with the same value. If this is the first time that value occurred, then the tuples from left relation with that value will be removed from the output. Since the old value no longer exists, if no other tuples with that value also exists in the right relation then the tuples from the left relation containing that value need to appear in the output (because, now there are no matching tuples in the right). Both the above steps require information not within the current tuple, and obtaining such information is not efficient in the current framework. Bringing together all matching tuples from right relation for a single left tuple will require a sort or a hash. Both are extremely expensive, and performing them once per left tuple is impossible.

Remember that all the predicates are calculated after the Gibbs sampler, and the Gibbs queue (Section 2.7.1), that provides tuples to Gibbs sampler, does so only on one seed identifier (Section 2.4) at a time. Therefore, we need a change in the current framework for the execution of an anti-join predicate. Sorting all tuples in Gibbs queue is an option. With a sort on the value we can group the required tuples (with different seed identifiers) together. Though sorting solves the problem, it is not efficient since it
needs to be done once for each tuple. A single sort itself is very expensive, therefore performing a sort for each tuple will be extremely time consuming.

**STOC. CUSTOMERS**

<table>
<thead>
<tr>
<th>SEED</th>
<th>STOC. ARRAY CUST ID</th>
<th>BITSTRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>21 27 27 23</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>S2</td>
<td>23 21 23 21</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>S3</td>
<td>24 24 54 24</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>S4</td>
<td>64 24 84 64</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>S5</td>
<td>43 43 43 43</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

Figure 6-2. Example: Stochastic table

Consider the table Customers in Figure 6-1. Assume that the CUST_ID attribute is stochastic. The new table with the stochastic array attribute is shown in Figure 6-2. Figure 6-3 shows the stochastic table after the Split operation, along with the left table in the anti-join. The tuple with CUST_ID 21 in Orders matches with all two tuples in Stoc.Customers, tuples 1 and 4 with seeds S1 and S2 respectively. The actual anti-join predicate (like all other predicates) is computed after the Gibbs sampler step. So in order to know if the tuple with CUST_ID 21 in Orders appears in the output, we need both the tuples from Stoc.Customers and which of them are active for a given DB instance. But the Gibbs queue only provides groups of tuples with same seed together. So in the current system it is impossible to execute an anti-join predicate. We need some non trivial changes in the system for this to work.

In the next section we discuss two new attribute formats and how they will facilitate the anti-join in MCDB-R. After that we explain the modifications required in the system so that the anti-join will work in case of extreme sample problem. In Section 6.4 we discuss briefly how to extend both formats if more than one anti-join is present in a query. Finally in Section 6.5 we evaluate both formats described below for an anti-join execution in various types of datasets.
As discussed previously, to execute an anti-join predicate we need all matching tuples from the right relation. In MCDB-R we apply the predicate only after the Gibbs sampling step, hence all the matching tuples from right should be collected to run the anti-join. This is complicated in the current framework because at the Gibbs sampler we only have tuples containing a particular seed identifier. The current interface of Gibbs queue does not allow retrieving tuples not related through a seed identifier. Adding a method to perform such operation in Gibbs queue will be very expensive. Such an operation will require a sort or hash every time we want to apply an anti-join predicate. In this section we discuss two new attribute formats that can get around this problem. We also explain how these new attribute types can facilitate the anti-join operation.

### 6.2.1 Long Format

In this format we store all the necessary information from the right relation in each tuple from the left relation. During the Anti-join operation we add the seed and bitstring.
attributes from all matching tuples in R to the tuple in L. The seed is used by Gibbs queue to pull it up when the corresponding seed identifier is being processed. Figure 6-4 shows the structure of this composite attribute. The new attribute type is called AJ OBJECT (AJ for Anti-Join). It first keeps the number of seed and bitstring attribute pairs it contains (same as the number of matches in R), and then the list of seeds followed by the list of bitstrings.

| 3, SEED−1, SEED−2, SEED−3, BITSTRING−1, BITSTRING−2, BITSTRING−3 |

Figure 6-4. Composite Attribute: Long format

The BuildSeeds() method, which constructs the sorted array of seed identifiers for each tuple needs to be modified to incorporate the new attribute. This sorted array is later used by Gibbs queue to retrieve the tuples in the order of their seed identifiers. Figure 6-5 shows the new Gibbs tuples for the relations from Figure 6-2 after the anti-join operation. The new composite attribute is located at the end of the output tuples. The tuple with CUST ID 21 from Orders matches with two tuples (first and fourth) in the table Stoc.Customers. Therefore, the AJ OBJECT in the first tuple of output table Orders_AJ in Figure 6-5, stores two seed attributes and two bitstrings. Similarly, fifth tuple in Orders does not match with any tuple in Stoc.Customers, therefore its AJ OBJECT does not have any seed or bitstring attributes.

Processing of these tuples at the Tail sampler is explained below. For simplicity we are assuming that there are no operators in between Anti-join and Tail sampler. Also assume that we have only one DB instance, and initially all the DB instances are pointing to the first sample in the stream. At first, the Gibbs queue returns the fifth tuple and since it does not have any seed attributes it is sent to the aggregate without any extra processing. In the next step, tuples with seed S1 are returned, i.e. first two tuples. Tail sampler will try to replace the current sample for DB instance 1 (DB1). Lets say that the Advance() will move the DB1 sample from iteration number 1 to iteration number 2.
The bit changed from 1 to 0, therefore DB1 of S1 now has a value 23 instead of a 21.
Now the validity of tuple 1 is checked through the anti-join predicate. Since there is still
a valid 21 from the composite attribute due to DB1 of S2, the status of anti-join predicate
on this tuple does not change. So first tuple does not contribute to the aggregate. For
second tuple both bitstrings have a 0 at first iteration. The anti-join predicate will return
a true since there are no matches. Hence, initially the second tuple will appear in the
output and contribute to the aggregate. After the \texttt{Advance()}, the DB1 of S1 is assigned
23. The bitstring of S1 is 1 at second iteration. Therefore, the anti-join predicate will
return a false, due to the existence of a match in right relation. So, after an \texttt{Advance()} on
S1, the aggregate value changes. More samples are processed if the new aggregate
does not satisfy the constraint on its value, and the rejection algorithm proceeds further.

\textbf{Possibilities of overflow:} As seen in the above example, using the format is
fairly sample. We need to create a new data type and make minor modifications to
\texttt{BuildSeeds()} method and \texttt{Tail} sampler operator. Though this method works well in the
above example, it makes an inherent assumption that the new tuple, storing numerous
seed and bitstring attributes, will fit in a single page. The assumption holds, if, for a left
tuple the matching tuples from right relation are less. The tuple can easily exceed the
page size if there are too many matching tuples in right relation. Both the seed and
bitstring attributes are big. Bitstrings grow arbitrarily large based on the size of sample array. A sample array of size 10,000 will result in a bitstring of size 1250 bytes. With a page size of 1 MB, we can only hold close to 800 of such bitstrings. The size of seed attribute depends on the number of DB instances, $num_{db}$. If we have 100 DB instances, the seed is approximately 800 bytes. For a 1 MB page size, we can hold close to 500 of these seed and bitstring attribute pairs.

The attributes on which anti-joins are performed are categorical. In general the domain size of categorical attributes is small. Examples of such attributes are Location (Country, Zip code etc), Department, Component etc,. Small domain sizes and large relation sizes generally results in large number of matches. So the long format is not always appropriate.

### 6.2.2 Short Format

![image]

Figure 6-6. Composite Attribute: Short format

As explained above, creating huge tuples which are bigger than the system page size should be avoided whenever possible. Here we discuss the short format for $AJ OBJECT$ which reduces the attribute size. As explained before, the contributing factors for large tuple sizes are both seed and bitstring attributes. So the new format aims to avoid storing multiple seed and bitstring attributes in a single tuple. Instead, it stores only the seed identifiers of the matching tuples from right relation. We also store the parts of the bitstrings which are currently assigned to the DB instances. The size of these bitstring parts is the number of DB instances multiplied by the number of seed identifiers stores in the $AJ OBJECT$, and is very small. The right tuples themselves are put in the Gibbs queue, so that their seed and bitstring attributes are accessed whenever necessary. The maximum number of tuples inserted in the Gibbs queue with this solution will be the sum of the sizes of both the relations in the anti-join.
<table>
<thead>
<tr>
<th>ORD ID</th>
<th>CUST ID</th>
<th>AMOUNT</th>
<th>AJ OBJECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
<td>12</td>
<td>#2, 1, 2, 11</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>22</td>
<td>#2, 1, 2, 00</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td>21</td>
<td>#2, 3, 4, 10</td>
</tr>
<tr>
<td>4</td>
<td>44</td>
<td>25</td>
<td>#2, 3, 0</td>
</tr>
<tr>
<td>5</td>
<td>55</td>
<td>42</td>
<td>#0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SEED</th>
<th>STOC. SINGLE</th>
<th>CUST ID</th>
<th>BITSTRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>21</td>
<td>0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>S1</td>
<td>23</td>
<td>0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>S2</td>
<td>21</td>
<td>0</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>S2</td>
<td>23</td>
<td>0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>S3</td>
<td>24</td>
<td>1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>S3</td>
<td>54</td>
<td>0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>S4</td>
<td>24</td>
<td>0</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>S5</td>
<td>43</td>
<td>1</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

Figure 6-7. Short Format: Tuples after Anti-join

The resulting new AJ OBJECT can be seen in Figure 6-6. The size of this attribute is 33 bytes. In comparison the size of long format shown in Figure 6-5 for the sample attribute is close to 6,000 bytes (assuming the number of DB instances $num_{db}$ is 100 and size of sample array $num_{mc}$ is 1000). For the small format in this example, we also insert three additional tuples in the Gibbs queue. These three tuples have the seeds and bitstrings. In Figure 6-7, the first five tuples are from the left relation. These five tuples have their AJ OBJECT in the short format. In tuple 1, the AJ OBJECT has 2 seed identifiers – 1 and 2. Next 8 tuples are from right relation, and they are inserted into Gibbs queue directly (more on this later).

The difficultly with short format when compared to the long format is that it results in more changes in the MCDB-R system. As explained in Chapter 2, the execution flow assumes all tuples have the same schema. So sending the tuples from right relation
additionally into the output will result in tuples with different schemas sent to next
operator. To get around this issue we could create a super relation with attributes from
both. This solution still requires each operator in MCDB-R to differentiate between these
two tuples. Each operator needs necessary modifications, and so this solution is not
desirable. A cleaner solution is to send the tuples from right relation R to the Gibbs
queue directly. This method only requires changes to the anti-join operator. Though
this method is fairly simple for a single anti-join, it requires more complicated schemes
when there are cascading anti-joins in the query plan. A generic method that will work
for queries with cascading anti-joins is discussed later in Section 6.4. Since we have
tuples from both left and right relations in the Gibbs queue, the number of tuples it has
is high. A big drawback of using small format is the requirement of very high number of
remove/reinserts in the Gibbs queue. Each reinsert could potentially involve hundreds of
tuples and can slow down the execution.

6.3 Discussion on Extreme Samples

A sample is called extreme if the probability of its occurrence is very low. Replacing
an extreme sample during the rejection algorithm can be difficult. A solution to this
problem is discussed in Chapter 5. However, that solution only works for queries with
out an anti-join operation. In this section we briefly discuss how to extend the solution in
Chapter 5 to work even for queries with an anti-join.

This problem of replacing an extreme sample is solved by running the query plan
separately for just for finding that sample. After the sample is found, only then the
regular execution restarts. The modified query run has two steps. The first step is
filtering out unnecessary tuples from the execution in order to improve the efficiency
of the run. Filtering process will be different when there is a stochastic attribute on the
right side of the anti-join. There are multiple seed identifiers from the same relation
that produced the extreme sample, but not due to a self-join. From these multiple seed
identifiers, we call the one that generated the extreme sample as primary seed identifier.
To recreate the composite attribute we need all seed identifiers present in the tuple. The new samples generated for the primary seed identifier can have a new value not seen before. These new values can match with other new tuples. We can not filter out any seed identifiers, but for all identifiers except the primary seed identifier we need to generate only the active samples. The unnecessary tuples can be filtered out after the Anti-join operator. The Anti-join operator must be modified to handle the fragmented tuples for the primary seed identifier. This again is similar to how a self-join is modified (Section 5.3).

6.4 For Queries with Multiple Anti-Joins

Some queries can have cascading anti-joins. Two anti-joins are called cascading if the output of one anti-join is given as a right input to another anti-join. The current composite attribute formats work only for queries with non cascading anti-joins. The creation of the composite attribute should be recursive for these queries with cascading anti-joins. A cascading anti-join at second level, one whose right input is from another anti-join, stores the AJ OBJECT in the right tuples instead of just seed and bitstring attributes. The function (in Tail sampler) to retrieve the current active DB instance from the composite attribute would also need changes. This function also needs to work recursively on the composite attribute. These ideas need more work and are not implemented in the system.

6.5 Experiments

In this section we benchmark both of our methods discussed above – Long format and Short format. Our goal is to compare their efficiency and scalability for different settings. The following are the questions we like to answer:

1. How does both formats work when large number of Seed attributes need to be stored?
2. Which of them is better on large datasets?
**Experimental Setup:** We use the TPC-H benchmark data generator [17]. The workstation used for the experiments has 32 GB of RAM, 4 720 GB Dell hard disks and 8 processing cores of 3.2 GHz each distributed over 2 chips. The system runs a 64-bit Ubuntu operating system, version 11.04. We implemented our solution in the prototype MCDB-R system. We use the query Q5 explained below. See Appendix for complete SQL statements for Q5. Query Q5 is a modified version of query Q3 discussed before in Section 5.6.

**Query Q5 (Modified Q3):** Here we calculate the error in calculating total revenue due to errors in the data integration process. The schema is modified for a simulating data cleaning process. The customer information in the orders relation is assumed to have errors. A new stochastic relation is created to incorporate these errors. This relation has the actual customer key and a list of possible customer keys, and a probability associated with each possible customer key. The query Q4 joins customer table with orders, and an order contributes to the total revenue only if it finds a valid key to match in customers. In the current query, we calculate the orders which failed to contribute to the revenue because of the non existence of the customer information in the customer table.

To control the number of tuples from customer that join with each order tuple, we regulate how the possible customer keys are generated in the new relation. We divide customer key values into non overlapping groups. For each key in a given group the list of possible customer keys are from that same group. The probability of each possible key is same, and is \( \frac{1}{m} \), where \( m \) is the size of the group in which the actual customer key belongs to. The size of all groups is same, and this size \( m \) is the number of tuples we want from customer relation to join with a single tuple in orders table. So, if we want the composite attribute to have 300 seeds, then we need 300 tuples from the customer relation to join with that tuple. This can only happen if the same value is generated in 300 tuples in customers and hence the group size \( m \) should be 300. The
Table 6-1. Performance with varying seeds per tuple (in minutes)

<table>
<thead>
<tr>
<th>Num. of Seeds per Tuple</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long format</td>
<td>2.7</td>
<td>3.0</td>
<td>3.8</td>
<td>7.3</td>
<td>19.8</td>
<td>35</td>
</tr>
<tr>
<td>Short format</td>
<td>2.3</td>
<td>2.5</td>
<td>3.6</td>
<td>7</td>
<td>13</td>
<td>19.3</td>
</tr>
</tbody>
</table>

equal probability for each possible value will ensure that there is a good chance that each value will be generated in the sample array.

Given this setup, there are five different variables we need to know to understand the results:

1. **numSeedIds**: The number of seed identifiers processed during the query execution.

2. **num\(_{db}\)**: Number of DB instances for the query. This is set to 100 for all tests in this section.

3. **SeedsPerTuple**: The average number of seed attributes stored in each AJ OBJECT.

4. **gibbsIters**: Number of iterations for Gibbs Looper. This value is 5, unless specified otherwise.

5. **eliteDBPercent**: Number of DB instances retained at the end of each Gibbs iteration. The deleted DB instances are cloned from the retained elite instances. We have this value at 50% for all the tests.

We use two tests for evaluation. Test 1 is for comparing the performance of both formats with increasing AJ OBJECT size. In Test 2, we compare the performance of the two formats on varying sizes of the composite attribute formed, and also by changing the number of input tuples (numSeedIds). Both tests use query Q5. The variables for each of the tests are as follows:

1. **Varying SeedsPerTuple**: For this test, numSeedIds is set to 1000. Value of SeedsPerTuple is varied from 10 to 300. num\(_{db}\) is set to 100, gibbsIters to 5, and eliteDBPercent to 50%.

2. **Performance on large datasets**: For this test, numSeedIds is set to 100,000 and 15,000 for dataset 1 and dataset 2 respectively. Value of SeedsPerTuple is 10 and 300 for the datasets 1 and 2 respectively. num\(_{db}\) is set to 100, gibbsIters to 5, and eliteDBPercent to 50%.
Table 6-2. Performance on large datasets (in minutes)

<table>
<thead>
<tr>
<th></th>
<th>Dataset 1</th>
<th>Dataset 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num. of Tuples</td>
<td>100,000</td>
<td>15,000</td>
</tr>
<tr>
<td>Seeds per Tuple</td>
<td>10</td>
<td>300</td>
</tr>
<tr>
<td>Long format</td>
<td>30.8</td>
<td>552</td>
</tr>
<tr>
<td>Short format</td>
<td>54.6</td>
<td>258.3</td>
</tr>
</tbody>
</table>

**Results:** Results for Test 1 are given in Table 6-1. The table shows the performance of both formats with varying number of seed attributes stored in the composite attributes, the \texttt{AJ OBJECT}. The number of seed attributes per tuple is the group size \( m \) as described in the query description. All the values listed are the number of minutes taken for the query execution. Table 6-2 shows the performance results for Test 2. The query is run on two datasets. The execution times are shown under Long format, and Short format columns.

**Discussion:** Test 1: As can be seen in the Table 6-1, for smaller number of seed attributes per tuple (till 100) both solutions perform almost similarly, with Short format taking a slight edge. For the other two cases (200 and 300 seed attributes per tuple) there is a clear difference between the performance. Here the experimental results are as expected. The Long format, with larger tuple sizes reads/writes more data from/to disk. The execution time increases with increasing number of seed attributes. The reasons here are two fold. For Long format, the size of tuples becomes larger and for Short format, the number of tuples in the Gibbs queue grows. When \( m \), the number of seed attributes per tuple is 300, the size of tuple in the Gibbs queue for Long format is approximately 260 KB. For Short format, though the tuple size is very less, the number of tuples in Gibbs queue is around 270,900. The second reason is the number of tuples processed in a single Gibbs iteration. Since each tuple has more seed attributes now, the number of tuples pulled out for each seed attribute are more and so are the reinserts into the Gibbs queue. This will increase the execution time considerably.
Test 2: The results are shown in Table 6-2. First dataset has 100,000 tuples and 10 customers in each group. Therefore, the number of seed attributes in the composite attribute will be 10. Surprisingly Long format performs better on dataset 1. This can be attributed to Short format being slowed down by all the extra tuples in the Gibbs queue. For this dataset, for Short format the Gibbs queue contains 1,100,000 tuples, whereas Long format contains only 100,000 tuples. The second dataset has 15,000 tuples and 300 customers in each group. This results in 4,515,000 tuples in the Gibbs queue for Short format. Even with so many tuples, it performs better than Long format, because of its lower disk usage. For this query, each tuple in the Gibbs queue is removed and reinserted until all seed attributes in that tuple are exhausted. Since the tuples are massive (approximately 260 KB) for Long format, multiple remove and reinserts are more expensive.

Conclusion: After looking at the results from both tests, the Short format seems more useful. Though we can not say it is always better, in most results it outperforms the Long format. The most important factor in recommending Short format is the inability of Long format to store a large number of seed attributes.
CHAPTER 7
CONCLUSION

We presented two methods to improve the performance of MCDB-R system. The first method is to serialize and de-serialize the sample generating process. We implemented and tested the idea in the prototype system. As shown in the experiments in Section 4.7, for many types of VG functions the performance benefits are great.

Next, we presented an efficient technique to replace extreme samples. We provided a two step process; filtering out unnecessary tuples and fragmenting the very large tuple bundle. The implementation of this process resulted in significant performance benefits, an improvement factor of 2 or better in many cases, as shown in Section 5.6.

Finally, we looked at how to incorporate anti-join operator in MCDB-R. We gave two solutions which work for queries with a single anti-join operation. Both solutions are orthogonal in how they generate tuple bundles at the anti-join operator. We will look at extending the work to facilitate multiple anti-join operators in a single query plan.
APPENDIX: QUERIES

Query Q1

CREATE VIEW params AS

SELECT 2.0 AS p0shape, 1.333*AVG(l_extendedprice *(1.0-l_discount)) AS p0scale, 2.0 AS d0shape,
   4.0*AVG(l_quantity) AS d0scale, l_partkey AS p_partkey
FROM lineitem l
GROUP BY l_partkey

CREATE TABLE demands (new_dmnd, old_dmnd,
   old_prc, new_prc, nd_partkey, nd_suppkey) AS
FOR EACH l IN (SELECT * FROM lineitem, orders
   WHERE l_orderkey=o_orderkey AND
eyr(o_orderdate)=1995)
   WITH new_dmnd AS Bayesian (
      (SELECT p0shape, p0scale, d0shape, d0scale
         FROM params
         WHERE l_partkey = p_partkey)
      (VALUES (l_quantity, l_extendedprice*(1.0-
l_discount))/l_quantity, l_extendedprice*
         1.05*(1.0-l_discount)/l_quantity))
      SELECT nd.value, l_quantity, l_extendedprice*
         (1.0-l_discount))/ l_quantity, 1.05*
         l_extendedprice*(1.0-l_discount)/l_quantity,
         l_partkey, l_suppkey
FROM new_dmnd nd

SELECT SUM (new_prf-old_prf) AS totalProfit
FROM (
SELECT
    new_dmnd*(new prc-ps supplycost) AS new_prf
    old_dmnd*(old prc-ps supplycost) AS old_prf
FROM partsupp, demands
WHERE ps_partkey=nd_partkey AND
    ps_suppkey=nd_suppkey)
WITH RESULTDISTRIBUTION MONTECARLO(100)
DOMAIN totalProfit <= QUANTILE(0.001)
FREQUENCYTABLE totalProfit

Query Q2

CREATE TABLE random_ord (o_orderkey, o_yr, o_tot) AS
    FOR EACH o IN (SELECT * FROM orders)
    WITH VAL AS Normal (VALUES(o_mean, o_var))
    SELECT o.o_orderkey, year(o.o_orderdate), v.value
    FROM VAL v
SELECT SUM(val) as totalLoss
FROM random_ord, lineitem
WHERE o_orderkey = l_orderkey AND
    (o_yr = 1994 OR o_yr = 1995)
SELECT SUM(grpsize * o_mean) AS mean,
    SUM(grpsize * grpsize * o var) AS var FROM
(SELECT o_mean, o_var, COUNT(*) AS grpsize
FROM orders, lineitem
WHERE year(o_orderdate) in (1994, 1995)
AND o_orderkey = l_orderkey
GROUP BY o_orderkey, o_mean, o_var)
WITH RESULTDISTRIBUTION MONTECARLO(100)
DOMAIN totalLoss >= QUANTILE(0.999)

**Query Q3**

CREATE VIEW from_japan AS

SELECT c_custkey
FROM customer, nation
WHERE n_nationkey = c_nationkey AND n_name = 'JAPAN'

CREATE TABLE fixed_cust AS

FOR EACH o IN orders

WITH newcustkey AS DiscreteChoice(
    SELECT custkey, probability
    FROM error_custkey
    WHERE old_custkey = o_custkey)

SELECT value AS o_newcustkey, o_orderkey
FROM newcustkey

SELECT SUM(l_extendedprice * (1.0 - l_discount)) as revJapan
FROM lineitem, fixed_cust, from_japan
WHERE l_orderkey = o_orderkey AND o_newcustkey = c_custkey
WITH RESULTDISTRIBUTION MONTECARLO(100)
DOMAIN revJapan >= QUANTILE(0.999)

**Query Q4**

CREATE TABLE actual_ship AS

FOR EACH o IN order_ship

WITH newshipdays AS Poisson(
    SELECT a.AvgShipDays
    FROM avg_ship a
    WHERE a.ShipDays = o.ShipDays)

SELECT ShipDays, AvgShipDays AS ExpShipDays
FROM newshipdays
SELECT SUM(s.Penalty) as totalPenalty
FROM actual_ship a, ship_penalty s
WITH RESULTDISTRIBUTION MONTECARLO(100)
DOMAIN MaxPenalty >= QUANTILE(0.99)

Query Q5
CREATE VIEW from_japan AS
SELECT c_custkey
FROM customer, nation
WHERE n_nationkey = c_nationkey AND n_name = 'JAPAN'
CREATE TABLE fixed_cust AS
FOR EACH o IN orders
WITH newcustkey AS DiscreteChoice(
    SELECT custkey, probability
    FROM error_custkey
    WHERE old_custkey = o_custkey)
SELECT value AS o_newcustkey, o_orderkey
FROM newcustkey
SELECT SUM(o_totalprice) as totalLost
FROM orders
WHERE o_newcustkey NOT IN
    (SELECT o_custkey
    FROM fixed_cust)
WITH RESULTDISTRIBUTION MONTECARLO(100)
DOMAIN totalLost >= QUANTILE(0.999)
REFERENCES


BIOGRAPHICAL SKETCH

Ravi Jampani is a PhD student at the University of Florida, Gainesville. He is advised by Dr. Chris Jermaine and Dr. Alin Dobra in the broad research topic of database management systems. In databases, his specific interests are in query processing and indexing. Apart from databases, he is also interested in the fields of natural language processing, multi agent systems, data structures and algorithms. Prior to joining University of Florida, Ravi received Bachelor of Technology (2003) and Master of Science (2005) degrees from the International Institute of Information Technology (IIIT), Hyderabad, India.