RICE UNIVERSITY

Query Processing and Optimization for Database Stochastic Analytics

by

Luis Leopoldo Perez

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Christopher M. Jermaine, Chair
Associate Professor of Computer Science

T.S. Eugene Ng
Associate Professor of Computer Science and Electrical and Computer Engineering

Peter J. Varman
Professor of Electrical and Computer Engineering and Computer Science

Houston, Texas
December, 2014
The application of relational database systems to analytical processing has been an active area of research for about two decades, motivated by constant surges in the scale of the data and in the complexity of the analysis tasks. Simultaneously, stochastic techniques have become commonplace in large-scale data analytics. This work is concerned with the application of relational database systems to support stochastic analytical tasks, particularly with the query evaluation and optimization phases.

In this work, three problems are addressed in the context of MCDB/SimSQL, a relational database system for uncertain data management and analytics. The first contribution is a set of efficient techniques for evaluating queries that require satisfying a probability threshold, such as “Which pending orders are estimated to be processed and shipped by the end of the month, with a probability of at least 95%?” where the processing and shipment times of each order are generated by an arbitrary stochastic process. Results show that these techniques make sensible use of resources, weeding out data elements that require relatively few samples during the early stages of query evaluation.

The second problem is concerned with recycling the materialized intermediate results of a query to optimize other queries in the future. Taking the assumption
that a history of past queries provides an accurate picture of the workload, I describe techniques for query optimization that evaluate the costs and benefits of materializing intermediate results, with the objective of minimizing the hypothetical costs of future queries, subject to constraints on disk space. Results show a substantial improvement over conventional query caching techniques in workload and average query execution time.

Finally, this work addresses the problem of evaluating queries for stochastic generative models, specified in a high level notation that treats random variables as first-class objects and allows operations with structured objects such as vectors and matrices. I describe a notation that, relying on the syntax of comprehensions, provides a language for denoting generative models and guarantees correspondence with relational algebra expressions, and techniques for translating a model into a database schema and set of relational queries.
Acknowledgments

I would like to thank my advisor, Prof. Chris Jermaine, for all the support, encouragement and patience that he has extended to me since I started working under his supervision at the University of Florida. His particular approach to teaching the skills related to solving research problems allowed me to work on many interesting and relevant projects, while slowly weaving the contents of this thesis.

My gratitude goes to the members of my dissertation committee, Prof. T.S. Eugene Ng and Prof. Peter Varman, for their invaluable suggestions and comments. I would also like to thank Prof. Alin Dobra and Prof. Manuel Bermúdez at the University of Florida, and Prof. Ignacio Trejos-Zelaya at CENFOTEC Costa Rica, for their encouragement during the beginning of my graduate studies.

In no particular order, I would like to thank some of my friends and collaborators at Rice University: Niketan Pansare, Sağnak Taşırlar, Zhuhua Cai, Jacob Zekai Gao, Shangyu Luo, Dragoş Sbîrlea, Ioan Şucan, Florin Dinu, Anna Drummond, Risa Myers, Zografoula Vagena, and Deian Tabakov.

I am immensely grateful to my parents, for placing my education above their personal comfort and preferences; to my siblings, for their support and encouragement; and to Betty, for always being there.
Contents

Abstract ii
Acknowledgments iv
List of Illustrations viii
List of Tables xi

1 Introduction 1
1.1 Evaluation of Probability Threshold Queries 3
1.2 History-aware Query Optimization 5
1.3 Natural Specification of Stochastic Models 7

2 Preliminaries 10
2.1 Related Work 10
2.2 Background 23
2.2.1 Sequential Hypothesis Testing 23
2.2.2 Cost-based Query Optimization 26
2.2.3 Gibbs Sampling for Generative Models 31
2.3 Overview of MCDB/SimSQL 35
2.3.1 The MCDB Data Model 41
2.3.2 VG Functions and Stochastic Table Creation 42
2.3.3 Query Evaluation 45
2.3.4 MCMC Simulation with SimSQL 48

3 Evaluation of Probability Threshold Queries 55
3.1 Overview of Threshold Queries 56
3.1.1 Statistical Foundations ........................................... 59
3.2 Query Evaluation ...................................................... 60
  3.2.1 The Partition-and-Test Operation ............................. 63
  3.2.2 The Hypothesis Test operation ................................. 64
  3.2.3 Running Sequences of Test Blocks ............................. 65
  3.2.4 Processing Records with Lineage .............................. 68
  3.2.5 The Lineage Routing Operation ............................... 70
  3.2.6 Stochastic Considerations ...................................... 72
3.3 Experimental Evaluation .............................................. 74
  3.3.1 Queries Tested .................................................... 75
  3.3.2 Results ............................................................ 78
  3.3.3 Discussion ........................................................ 80

4 History-aware Query Optimization ................................. 85
  4.1 Platform Overview .................................................. 88
    4.1.1 Costing in the Optimizer .................................... 90
    4.1.2 The History Pool ............................................. 92
    4.1.3 The View Pool ............................................... 93
  4.2 Example Query Workload ........................................... 94
  4.3 Query Optimization in Detail .................................... 99
    4.3.1 View Enumeration and Matching ............................ 99
    4.3.2 Costing with the History Pool ............................... 103
  4.4 Updating the Pools ................................................ 108
    4.4.1 History Pool: Selecting Plans ............................. 108
    4.4.2 View Pool: Selecting Views ............................... 110
  4.5 Experimental Evaluation ......................................... 115
    4.5.1 Setup ......................................................... 116
    4.5.2 Benchmarks Run ............................................. 117
5 Natural Specification of Stochastic Models 127

5.1 Syntax and Semantics 132
  5.1.1 Data and Variable Element Definitions 133
  5.1.2 Model Statements 136

5.2 Overview of the BUDS Translator 143

5.3 Model Checking and Normalization 145
  5.3.1 Model Normalization 145
  5.3.2 Initialization and Sampling Order 148

5.4 Searching for Schemas and Query Implementations 150
  5.4.1 A Naive Solution 151
  5.4.2 Representing the search space 154
  5.4.3 Exploring the search space 164
  5.4.4 Sketch of the Search Strategy 167

5.5 Experimental Evaluation 168
  5.5.1 Models Tested 169
  5.5.2 Setup 171
  5.5.3 Results and Discussion 172

6 Conclusions 177

Appendix: Machine Learning Models in MCDB/SimSQL 179

A.1 The Bayesian Lasso 179
A.2 Latent Dirichlet Allocation 184
A.3 Gaussian Mixture Model 187

Bibliography 194
Illustrations

2.1 Likelihood ratio (a) and proportion of successful trials (b) as a function of the number of values observed by a SPRT. . . . . . . . . . 25
2.2 Equivalent query plans generated by the query optimizer. . . . . . . 28
2.3 Graphical representation of a simple linear regression generative model. 32
2.4 Observed values for the distribution parameters across 30 generations of a Gibbs sampler. . . . . . . . . . . . . . . . . . . . . . . . . 35
2.5 Workflow of query evaluation in MCDB. . . . . . . . . . . . . . . . 36
2.6 Sampling linear demand curves from a stochastic model. . . . . . . 37
2.7 Example output query in MCDB, containing a stochastic attribute with $N = 3$ possible worlds. . . . . . . . . . . . . . . . . . . . 40
2.8 Compact representation of tuple bundles with $N = 3$ possible database instances, before and after the application of a selection operation. . . . . . . . . . . . . . . . . . . . . 42
2.9 Query plans for instantiating MCDB stochastic tables. . . . . . . . 45
2.10 Dependency graph for updating the chain of stochastic tables for the linear regression Gibbs sampler, iterations $i = 0, 1, 2$ . . . . . . . . . . 52
2.11 Two possible frame cuts for the linear regression Gibbs sampler, iterations $i = 0, 1$ . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

3.1 Query plans for hypothesis testing with “standard MCDB/SimSQL” and threshold operators. . . . . . . . . . . . . . . . . . . . . . . . . 62
3.2 Query plan with multiple hypothesis test blocks. . . . . . . . . . . . 67
3.3 Application of the lineage routing operation in a threshold query. . . 71
3.4 Number of partitions (a) and wall-clock running time in seconds (b) per test block, for Query 3.1. . . . . . . . . . . . . . . . . . . . . . . . 79
3.5 Number of partitions (a) and wall-clock running time in seconds (b) per test block, for Query 3.2. . . . . . . . . . . . . . . . . . . . . . . . 80
3.6 Number of partitions (a) and wall-clock running time in seconds (b) per test block, for Query 3.3. . . . . . . . . . . . . . . . . . . . . . . . 81
3.7 Number of partitions (a) and wall-clock running time in seconds (b) per test block, for Query 3.4. . . . . . . . . . . . . . . . . . . . . . . . 82
3.8 Number of partitions (a) and wall-clock running time in seconds (b) per test block, for Query 3.5. . . . . . . . . . . . . . . . . . . . . . . . 83

4.1 Architecture of the Hawc query optimization framework. . . . . . . 88
4.2 Possible plans for Query 4.1 ($Q_1$). . . . . . . . . . . . . . . . . . . . 96
4.3 Possible plans for Query 4.2 ($Q_2$). . . . . . . . . . . . . . . . . . . . 97
4.4 Running time (HH) as a function of the fraction of executed queries in workload W1. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120
4.5 Running time (HH) as a function of the fraction of executed queries in workload W2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
4.6 Running time (HH) as a function of the fraction of executed queries in workload W3. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
4.7 Running time (HH) as a function of the fraction of executed queries in workload W4. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
4.8 Running time (HH) as a function of the fraction of executed queries in workload W5. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
4.9 Running time as a function of the fraction of executed queries with maximum view pool size $\frac{25}{3}$. . . . . . . . . . . . . . . . . . . . . . . 124
4.10 Running time as a function of the fraction of executed queries with maximum view pool size $\frac{S}{2}$. ............................................. 124
4.11 Running time as a function of the fraction of executed queries with maximum view pool size $\frac{S}{4}$. ............................................. 125

5.1 General architecture of the BUDS translator. ............................................. 143
5.2 Full variable dependency graph for the Bayesian Lasso. ......................... 147
5.3 Stochastic variable dependency graphs for the Bayesian Lasso (a) and LDA (b) models. ............................................. 149
5.4 Chain variable dependency graphs for the Bayesian Lasso (a) and LDA (b) models. ............................................. 151
5.5 Type lattice for the variable $X$ from the Bayesian Lasso. .................... 156
5.6 Expression graph for portions of the Bayesian Lasso. ......................... 159
3.1 Summary of experimental results for threshold queries 3.1-3.5, with a comparison between the End-Biased Test and the SPRT.

4.1 Cost estimates for the example query plans and views.
4.2 Total workload running time (HH:MM) without view pool space constraints.
4.3 Maximum total disk space occupied by the pool and total number of views created during the execution of each workload.
4.4 Query-wide comparison of benefits and costs for the query-based and history-based methods, with respect to the “no materialization” benchmark.
4.5 Total workload running time (HH:MM) with different pool space constraints and caching methods.

5.1 Number of lines of code and running times (MM:SS) for hand-written SQL and language implementations of the Lasso, LDA and GMM models.
Chapter 1

Introduction

Over the past two decades, much research has been devoted to the development of relational database systems for analytical processing tasks that enable the database as a tool for decision support, as opposed to transactional processing tasks—that is, the clerical data storage and retrieval tasks for which relational databases were originally conceived [1,2]. More recently, a large portion of the research work has been devoted to managing the constant increases in data size and scale [3,4], together with the emergence of more sophisticated forms of analysis [5-7]. Simultaneously, the application of stochastic models and analysis techniques to large data sets has become commonplace [8,9].

The main theme of this thesis is the application of stochastic modeling and analysis in relational databases. In particular, this thesis is concerned with the phases of query evaluation and optimization in a relational database system (RDBMS) that has been extended to carry out stochastic analytical tasks. To illustrate the usefulness of “pushing” this form of analysis into the database, a motivating example from [10] is used.

Example 1.1. An analyst at a retail company decides to use the contents of a relational database with information on past orders to apply the following what-if analysis: “What would our profits have been during the last 12 months, if we had raised our prices by 5%?” To this end, the analyst employs a Bayesian model of the
probability distribution of customer demand of a particular item after the hypothetical price increase has been observed. The parameters of the model, obtained from the database, are aggregate values linked to the price and demand of any given item. To apply the model, each past order is re-examined to determine its hypothetical size (i.e., the quantity of each item being ordered) at the increased price and, in consequence, the hypothetical revenue.

Carrying out the analysis described in Example 1.1 requires the application of Monte Carlo methods [11] to generate samples from the distribution of hypothetical order sizes, given the difficulty of obtaining an analytical solution to the distribution function for customer demand of a given item at a new price. In a conventional setting, this analysis would be implemented externally using a language such as R or MATLAB, and the parameters of the model would be extracted from the database and transformed into a suitable data format. This process is error-prone, difficult to maintain, and requires the allocation of system resources to re-parameterize the model every time the contents of the database are updated.

The Monte Carlo Database (MCDB) is a RDBMS for stochastic analytics and uncertain data management that addresses the above shortcomings [12]. The MCDB relational data model allows for the existence of conventional deterministic attributes, corresponding with single attribute values, alongside uncertain stochastic attributes that correspond to empirical distributions of possible attribute values generated by a user-defined stochastic model. The goal of MCDB is to provide a general framework to support a broad class of arbitrary stochastic models that, encapsulated as user-defined “variable generation” (VG) functions, generate data that can be queried using standard SQL statements.

SimSQL is an extension of MCDB aimed at providing support for Markov Chain
Monte Carlo (MCMC) simulation of stochastic generative models, which is a popular tool in Bayesian machine learning and data mining applications [13]. SimSQL allows tables containing stochastic attributes and VG function calls to be defined in mutually recursive fashion, so that a user may specify a “chain” or sequence for generating samples from the random variables that comprise a model.

The three problems discussed in this thesis are addressed in the context MCD-B/SimSQL. A brief description of these problems and the contributions of the thesis is presented in the remainder of this section.

1.1 Evaluation of Probability Threshold Queries

In MCDB/SimSQL, records in a database table can contain stochastic attributes. In contrast to the conventional deterministic attributes, a stochastic attribute within a record is associated with more than a single attribute value: it contains an empirical distribution with a sample of \( N \) possible attribute values, where \( N \) is a system-wide constant. The question of choosing an appropriate value of \( N \) is left to the end-user.

Consider the analysis from Example 1.1. Suppose a user writes a MCDB/SimSQL query to compute a summary of the results of the analysis as a table containing a stochastic attribute with the distribution of customer demand for each item at a 5% price increase. To draw inferences such as “Which items would have never been ordered, with 95% probability?” an end-user must apply appropriate statistical analyses on the output of MCDB/SimSQL as an external and separate process. This approach has several drawbacks:

1. The user must determine the appropriate statistical hypothesis test and provide an implementation that takes the data extracted from the output of a MCDB/SimSQL query.
2. Depending on the test being applied and the desired level of accuracy, the user must determine a proper value of $N$ to make the inference statistically significant. This process can be complex enough to become a separate analysis by itself, often requiring the generation of “pilot” samples, implying multiple runs of the MCDB/SimSQL query to generate more data.

3. Given the difficulty behind estimating $N$, it is highly likely that MCDB/SimSQL will be used to generate an incorrect amount of data—either not enough observations, which implies that the query must be repeated, or too many observations, which results in a waste of resources.

4. Moreover, it is often the case that many of the data elements being tested require smaller samples than others for the inference to be accurate. For example, in the query described above, a relatively low amount of observations should be required to provide a “negative” answer for those items that stay in very high demand after the price increase. Nonetheless, MCDB/SimSQL will generate $N$ observations for every item, which is likely to result in a waste of resources.

To address the above problems, I propose pushing statistical inference based on hypothesis testing into MCDB/SimSQL by introducing a novel category of queries, called threshold queries, that make it possible to specify the hypothesis to be tested, the level of granularity of the data elements being tested, and the confidence level of the test. Being tasked with applying the test on each data element allows MCDB/SimSQL to have exact knowledge of the size of the samples that need to be generated, guaranteeing the validity of the inference without wasting resources.
1.2 History-aware Query Optimization

In database environments wherein analytical processing tasks are the norm, materialized views are a common strategy for improving performance, as shown by the extensive body of research in “query caching” and database tuning methods [14]. A materialized view is, essentially, a physical structure that contains the result set of a given query sub-expression. The rationale behind using materialized views is that, given that the database schema of a data warehouse is often comprised of a relatively small amount of tables whose contents change infrequently, queries in a workload have a high amount of common sub-expressions that can be pre-computed and stored, therefore amortizing execution time in later queries [15].

Many analytical workloads, particularly those featuring statistical analysis (e.g., those that MCDB/SimSQL is designed to process), tend to feature iterative repetitions of queries with expensive sub-expressions over a data set that is largely constant. These workloads can be optimized to a high degree if the result sets of those constant sub-expressions are stored as materialized views. However, as the example below demonstrates, existing query caching and tuning techniques are not optimal for handling such scenarios.

Example 1.2. A relational data warehouse stores customer information as two separate tables: nation(nation_id,name) and customer(customer_id,balance, nation_id) where nation_id is a numerical identifier unique to each nation. An analyst issues a query to calculate the total account balance for all Japanese customers. A few minutes later, the analyst issues a similar query for all German customers, followed by another query for all American customers, and so on.

A conventional query optimizer is likely to recommend the following access path
(also known as *query plan*) for executing the queries from the workload described in Example 1.2: 1. Scan *nation*, filtering out any records that do not match the specified nation name; 2. Join this result with *customer* to obtain the set of customers with matching *nation_id*; and 3. Aggregate the account balance of those customers. A database runtime that supports query caching may create materialized views with the result set of any of these three operations, but such views would not be useful for optimizing other queries in the workload.

The problem with the above scenario is that the recommended plan is optimal with respect to a query executed in isolation, since the optimizer does not use any information about the workload. To address this, I propose a cost-based query optimization framework that makes use of a history of previously executed queries to guide optimization decisions. The basic premise is that such a history is a reliable picture of the needs of the workload. Therefore, a plan that is considered as sub-optimal for an isolated query may be preferred over the best plan, if the result set of the operations of the sub-optimal plan can be used to construct views with the potential to minimize the cost of future queries.

Under the query optimization framework I describe, the following sub-optimal plan may be preferable: 1. Join *nation* with *customer* so that each customer is associated with the name of the nation indicated by their *nation_id*; 2. Filter this result, returning only the customers that match the specified nation name; and 3. Aggregate the account balance of those customers. With this plan, future queries in the workload may recycle the results of the first operation to speed up execution. Hence, the query optimizer relies on the history to determine if the additional cost of executing this plan once is outweighed by the benefits it provides to future queries.
1.3 Natural Specification of Stochastic Models

Multivariate probability distributions are prevalent in contemporary stochastic models, particularly in the machine learning literature. In consequence, model variables are usually represented using matrices and vectors, and models are described in terms of matrix and vector operations. Concerning implementation, users prefer languages that provide native support for matrix types and linear algebra operations, such as R, MATLAB, or BUGS [16].

Representing vectors and matrices as tables in a relational database is a fairly straightforward task. However, because of the set-based nature of relational algebra operators, manipulating ordered structures such as arrays, matrices, graphs and lists is cumbersome and error-prone. The following example gives a rundown of these difficulties.

**Example 1.3.** The Lasso of Tibshirani [17] is a popular method for multidimensional linear regression that improves the quality of the regression model by discarding highly correlated, redundant dimensions. The Bayesian formulation of the Lasso, introduced by Park and Casella [18], models the data set as a response vector and a regressor matrix, requiring the use of Gibbs sampling (a Markov Chain Monte Carlo method) for estimating the regression model by repeatedly updating and drawing samples from the variables that comprise the model.

The Bayesian Lasso can be executed in MCDB/SimSQL. Following the relational model, the schema for the data set contains two separate tables, response(respID, respValue) and regressor(respID,regID,regValue), containing one record for each entry in the response vector or regressor matrix, respectively. The random variables are described using special purpose SQL syntax to create a set of mutually
recursive table definitions, with one table for each random variable in the model.

Implementing the Bayesian Lasso in MCDB/SimSQL requires approximately one hundred lines of code (see Appendix A.1 for the complete listing). The most difficult part of the implementation process is the translation of basic linear algebra operations, namely matrix and vector multiplications, into relational sub-queries containing joins and aggregates, often resulting in code that is difficult to read and maintain.

The lack of direct support for ordered structures and their difficult manipulation has been identified as a key reason behind the limited acceptance of relational databases in scientific applications [19]. We note that, although helpful, merely supporting matrix and vector attribute types in a system such as MCDB/SimSQL is not enough to simplify the process of specifying the stochastic model. In fact, adding such types increases the number of possible schemata that may be used to represent the same model, and the user is tasked with finding an appropriate schema for the size and constraints of the problem, which also determines how linear algebra operators are to be used.

To address these difficulties, I propose a high-level notation and language called BUDS for representing stochastic models in a natural, succinct way, so that an end-user can describe data structures in terms of matrices, vectors and sets, and specify how to sample each of the random variables in a model. Two particular conditions for this notation and language must be met: first, its expressive power must not exceed that of relational algebra; and, secondly, its data types must be independent from the relational schema representation. Therefore, I describe techniques for validating and translating models into an optimal schema and a set of relational algebra queries.
Outline of This Thesis

This thesis is organized as follows. Chapter 2 presents a survey of the related work and a description of some background concepts and methods that the rest of the thesis requires. Chapter 3 describes the solution to the problem of evaluating probability threshold queries. Chapter 4 discusses my approach to history-aware query optimization. Chapter 5 presents the notation and techniques for specifying and translating stochastic models. Chapter 6 concludes the thesis.
Chapter 2

Preliminaries

The three basic problems discussed in this thesis are connected to a variety of research areas, including, but not limited to, database systems architecture, database query optimization, large-scale distributed data processing and machine learning and data mining.

The purpose of this chapter is twofold: to present a review of the research literature that my work builds upon or is related to, and to provide a conceptual basis for the rest of the thesis, with a thorough description of the most relevant ideas from the literature. Thus, Section 2.1 presents the survey of the related work in the literature, Section 2.2 presents a more comprehensive review of some concepts from the literature, and Section 2.3 describes the MCDB/ SimSQL platform.

2.1 Related Work

Query Evaluation and Optimization in RDBMSs

Queries in relational databases are often specified using some derivative of the Structured Query Language [20], or SQL for short. The purpose of SQL is to allow queries in a relational database to be specified in declarative fashion—that is, as a logical description of what the result set of the query looks like, rather than an algorithmic description of how the result set is computed. Queries in SQL are expressed using SELECT statements that describe the attributes present in each record in the result
set, the input relations (also referred to as tables) that contribute records to the computation, and the logical conditions that each input record must satisfy in order to be part of the result set computation.

In order to be executed by the database engine, a SQL SELECT statement must be translated into a query plan (also known as access path) that contains an algorithmic, sequential description of the computations that must be carried out in order to obtain the result set of the query. In relational databases, plans are represented using relational algebra [21], which encapsulates computations into a few operations that take one or more relations as inputs and produce a single output relation.

Translating a SELECT statement into a valid relational algebra query plan that can be executed is a relatively straightforward task [22]. However, the resulting query plan is usually just one among the many semantically equivalent plans that can be generated for the same SQL statement. Thus, database engines carry out a query optimization stage so as to select the most efficient query plan for the query.

In general, query optimization strategies fall into two categories: rule-based optimization [23,24], wherein the optimizer inspects the query plan for the presence of particular patterns, replacing matching sub-plans with more efficient expressions, and cost-based optimization [25,26], which consists of iteratively generating equivalent query plans, calculating their individual “costs” and searching for a plan with minimal cost.

A crucial part of the query optimization process is deciding the order in which a sequence of relational join operations should be executed. Selinger, et al [25] noted that, although the cardinality of joining many relations is the same regardless of join order, the cost of those orders can be different, proposing a dynamic programming heuristic that computes the costs of joining subsets of relations and combines them to
construct the complete query tree. Nonetheless, the problem of finding the optimal join ordering has been shown to be NP-hard for general cost estimation functions [27]. Thus, several heuristic algorithms have been applied for finding near-optimal query plans in reasonable time. Ioannidis and Wong [28] introduced optimization algorithms using simulated annealing, which is a probabilistic hill climbing search algorithm. Swami and Gupta [29] proposed combinatorial heuristics based on iterative improvement for optimizing queries with large amounts of join operations. Bennett, et al [30] described a genetic algorithm that encodes query trees and iteratively combines and improves a population of possible solutions.

**Materialized Views and Query Caching**

A materialized view is a physical structure containing the result set of a given query expression or sub-expression, usually created with the purpose of speeding up queries that have overlapping, common sub-expressions [14, 15]. Relying on spare physical storage (usually disk space), materialized views allow for the amortization of the cost of workloads of queries by computing common query sub-expressions only once and storing their contents for later queries.

Materialized views are understood as derived relations—that is, as redundant tables whose contents can be computed by executing queries over the non-redundant tables in the database. The problem of answering queries using derived relations has been studied in depth in the database literature [31–34], and can be summarized as follows: “Given a query $Q$ and a set of derived relations $R$, generate a query plan that uses some of the relations in $R$ to compute the result set of $Q$ (or of $Q$’s sub-queries).” Achieving this requires searching $R$ for derived relations with algebraic definitions that are semantically equivalent to those of $Q$ or any of its sub-queries.
One of the classic papers in the area of materialized views and derived relations is by Goldstein and Larson [35], who describe scalable techniques and data structures for searching a large pool of materialized views derived from Select-Project-Join queries, so as to match their definitions efficiently with inbound queries in a data warehousing environment.

Query caching is closely related to methods for using materialized views, as derived relations containing the result sets of each query executed by the database engine are kept in a temporary cached and then used to process subsequent queries. The ADMS query optimizer [36] was the first solution to integrate the work in query execution via derived relations with the maintenance of a cache space containing intermediate query results, managed using traditional cache policies such as LRU.

Query caching techniques tend to be especially useful in data warehousing environments, where queries often have large amounts of common sub-expressions and the base, non-redundant tables in the database change rarely. The WATCHMAN [37] system was specifically designed for data warehousing, and relies on cache admission and eviction policies that protect the cache from taking in large and unpromising tables, improving the hit ratio of the derived relations in the cache. In a similar vein, the DynaMat [38] system was designed for efficient searching of materialized results by treating the data warehouse as a multi-dimensional repository in which the base data is organized among dimensions and the query cache as a pool of range fragments of the data dimensions, relying on a host of admission policies and “goodness” measures to maintain a high-quality cache.

Furthermore, query caching has been applied in distributed relational databases. Cache Investment [39] is a framework that integrates query optimization with workload analysis to manage data placement decisions in a distributed RDBMS. The goal
of Cache Investment is to analyze historical information continuously to identify useful tables and indices that satisfy a return-on-investment (ROI) criterion—that is, if the costs of caching a materialized result set and placing it on a particular node are smaller than the benefits obtained by other queries. To determine the benefits of a given data placement decisions, Cache Investment employs the query optimizer, using hypothetical information about the contents of the cache.

Recent research, such as the work of Ivanova, et al [40], focuses on architectures for query caching that perform the matching between individual physical operations and entries in the cache or pool during query execution, removing the need for substantive changes to the query optimizer. A recent application of the concept of query caching outside the realm of strictly relational databases can be found in ReStore [41], which aims at re-using the results of MapReduce jobs described in analytical query languages such as Pig [42].

Static analysis of query workloads was studied in the AutoAdmin project [43–45], with the goal of designing database administration and tuning tools with the capability of examining a query workload and obtaining an optimal configuration of materialized views and indices for improving the performance of the workload, subject to constrains on available spare disk space. Generating a configuration requires the execution of multiple queries and index creation statements—a process also known as an update phase. Extensions to the AutoAdmin static analysis approach that routinely analyze changes in the workload of queries that the database runtime executes and schedule any necessary update phases have also been studied [46].

Similar frameworks, such as Cache on Demand [47] and the MQT approach [48], are designed to aid the execution of batch-type query workloads. Prior to executing the queries, these systems analyze the workload in order to find out the best set of
intermediate query results to materialize and cache, and the order in which the sub-
queries in the workload should be executed so as to maximize the hit ratio of the
contents of the cache.

View Maintenance and Data Lineage

One of the drawbacks of using materialized views is that changing the contents of
the base relations from which a given view is derived can render the view useless for
answering queries, since the contents of the view would be inconsistent with the result
set of its defining query. A naive solution to this problem would involve re-computing
the contents of the view every time the contents of the base relations change, but
this would be unfeasible in any environment where changes to the base relations are
frequent or wide-ranging.

Much research work has been devoted to the problem of efficiently updating the
contents of a materialized view in the presence of changes to the non-redundant
relations of the database [49,50]. The goal is to examine the view definition and the
recent changes in the base relations to determine if the contents of the materialized
view are consistent with the contents of the database so that, in the case that it is
not consistent, find out the sequence of small update operations that must be applied
on the view.

A main concern in view maintenance is the tracking and management of the
lineage of the records present in a materialized view, a problem that has also been
studied in the field of data provenance [51]. The lineage of a given data object is the
set of objects that contributed to its existence. In a relational database, the lineage
$L_t$ of record $t$ would be the set of records from which $t$ was derived. Thus, if any of
the records in $L_t$ is modified, then $t$ must be modified.
In the context of view maintenance, two approaches to tracing the lineage of the records in a view can be delineated: explicit annotation [52,53], and so-called query inversion [54,55]. With explicit annotation, a unique tuple identifier is attached to every record $t$ in the database, alongside a lineage set containing the tuple identifiers of all the records that contributed to the creation of $t$. Explicit annotation implies that every relational operation has its own semantics for the computation of the lineage set of its output records. On the other hand, query inversion does not require the storage of any lineage information, but defines a method through which the lineage of a materialized view can be inferred by examining its contents and defining query.

**Analytical and Probabilistic Databases**

MCDB/SimSQL belongs to the category of analytical databases, particularly that of database systems whose capabilities have been extended so as to support statistical modeling tasks. Examples of such systems include MauveDB [56] and its successor FunctionDB [57], which provide a SQL interface for creating, managing and querying “model-based views” that describe regression and interpolation models over database tables, so that the model exists as a first-class object in the database.

There exist some similarities between MCDB/SimSQL and other systems for managing so-called probabilistic databases. Probabilistic databases extend traditional data models and their operations with abstractions that determine the degree of certainty that a given object exists in the database [58–60]. In general, data and query processing models for probabilistic databases tend to fall into two categories: models based on intensional semantics, and models based on extensional semantics [61]. In probabilistic databases based on intensional semantics, records in the database carry extensive event information from which probability values are computed, whereas
extensional semantics models rely solely on those probability values.

An example of intensional semantics is the Probabilistic Relational Algebra (PRA) introduced by Fuhr and Rölleke [62], which extends the relational database model so that each record in a table is associated with a “tuple weight” between 0 and 1 and a “tuple event” Boolean expression that, together, determine the probability of a record’s existence, assuming event independence between records. Dalvi and Suciu [63] present an extensional alternative that does not require explicit tuple event information for efficiently processing a wide-ranging class of probabilistic queries, and discuss categories of “hard” database queries that require the use of Monte Carlo simulation for computing tuple probabilities.

U-DBMS [64] is a probabilistic database management system that uses a data model that allows two general forms of uncertainty: attributes defined over an interval of non-decreasing possible values, and attributes defined by a probability distribution function. Query processing in U-DBMS adds two classes of queries to the relational model: entity-based queries, which return sets of objects with uncertain attributes, and value-based queries, which return numeric values computed from uncertain attributes.

Trio [65] is a system that extends the relational model so that records in the system (also known as “x-tuples”) can contain multiple alternative values for any given attribute (each with the probability of such value being true), uncertainty tags over the existence of a record in the database, and explicit lineage information identifying the tables and tuples from which the record was derived. Query processing in Trio allows for accessing, computing and querying confidence and lineage values.

The probabilistic database system MayBMS [66] is designed for incomplete information management using so-called probabilistic world-set decompositions (WSDs),
which denote the product of a series of probabilistic relations describing a finite number of possible worlds or instances of the database. Probabilistic relations in MayBMS store independent record fields in separate components of a WSD, while dependent components are stored together, with all components being associated with a probability value. MayBMS uses an extension of SQL with constructs for manipulating probabilities and data incompleteness.

Possible-worlds semantics, originally intended as a framework for artificial intelligence problems [67], is one of the most commonly employed conceptual tools in probabilistic databases. The intuition behind the possible-worlds model is to associate a database with uncertain values with a whole probability distribution of possible, independent database instances, each with different records, attribute values and individual probabilities. The generality of this model allows for treating a conventional, “deterministic” database as one in which all the possible instances have exactly the same records and attribute values.

Array Databases

Maier and Vance [19] argued that the lack of support for arrays and matrices in relational databases is a key reason behind their limited acceptance in scientific applications. Research in array databases is concerned with how array structures should fit within the data model and how the inclusion of these structures affects the query language.

One of the first approaches for supporting ordered structures in a database environment, presented by Seshandri, et al [68], introduced techniques and design principles for managing sequence-like structures –such as, for example, time series– as first-class objects on the same level as tables and views. Another seminal approach, the
RasDaMan system [69], introduced support for multi-dimensional arrays as second-class objects on the same level as data attributes.

Recent projects include SciDB [70], a database management system that aims at providing tools for scientific data analysis by providing a data model and query language that supports abstractions such as functions, matrices and vectors. In a similar vein, the SciQL system [71], provides a front-end with support for managing and querying ordered structures, while relying on a completely relational back-end.

The theoretical foundations for array database query languages were laid out by Libkin, et al [72]. Basic query processing techniques for array structures are present in the implementation of the AML query language [73]. A more recent proposal, the ArrayQL language, has been developed as a standard for array databases analogous to SQL, to be supported by both SciDB and SciQL.

Cornacchia, et al [74] introduced an array-based approach for information retrieval that relies on a relational database back-end, employing the RAM algebra of array operations [75] built around structural recursion and the syntax of comprehensions [76], which have been shown to be compatible with relational algebra [77].

**Distributed Analytical Processing**

The MCDB/SimSQL runtime is, in some ways, similar to that of systems such as Hive [78], HadoopDB [79] and Tenzing [80]. These systems provide an analytical relational database with SQL support and a back-end targeted to distributed environments where the database tables are spread throughout a cluster of commodity machines, using a distributed file system such as HDFS [81]. MCDB/SimSQL has such features, but contains extensions that allow for executing stochastic analytics and machine learning tasks in the database.
The popularization of MapReduce [82] for analytics and data mining, together with development of parallel and distributed versions of many popular machine learning algorithms such as [83–85] and the Apache Mahout library, laid the foundations for a new class of general data processing frameworks that combine a data model with a programming interface that allows for implementing machine learning algorithms in distributed environments. MCDB/SimSQL belongs to this category of systems. Other systems worth mentioning include GraphLab [86], Giraph [87], and Spark [88].

GraphLab is a framework for distributed analytics that uses a graph-based data model in which the vertices of the graph encapsulate processing tasks, while the edges encapsulate message passing between tasks. GraphLab’s C/C++ programming interface, built on top of MPI, allows for implementing data analysis tasks and models in scatter-gather fashion. Giraph uses a similar data model, but the programming interface is built in Java and relies on Apache Hadoop and HDFS for distributed task management and data storage, respectively.

The Spark framework is targeted towards the implementation of iterative machine learning algorithms in a fault-tolerant distributed environment similar to MapReduce. Spark relies on Mesos [89] for distributed task management, provides programming interfaces for the Python, Java and Scala language, and makes use of a data model based on structures known as Resilient Distributed Datasets (RDDs for short). RDDs are collections of objects partitioned across a cluster of machines that contain lineage information describing how the RDD was constructed in order to provide fault tolerance.
Probabilistic Programming

Probabilistic programming is part of a recent trend in artificial intelligence and programming languages research that focuses on the expressibility and computability of probabilistic models. A probabilistic program is a deterministic process with stochastic inputs, consisting of independent choices associated with probability distributions and a deterministic control flow over such choices [90].

The great variety of systems for probabilistic programming includes general programming languages, domain-specific languages for particular types of stochastic computations, and software libraries written for ordinary programming languages. General probabilistic programming languages are characterized by the inclusion of basic features for deterministic programming, often achieved by building on a previously existing programming language and extending it with support for stochastic variables and inputs and operations for conditioning variables with observations, inference for posterior probabilities and learning from data sets.

Church [91] is a probabilistic programming language that extends the Lisp language and its functional model of computation based on lambda calculus. Non-determinism is introduced in the language by way of stochastic primitive functions that generate random values according to a certain distribution. Church’s model of computation allows for the description of generative processes by treating program evaluation as sampling, including “querying” primitives for conditional sampling, schemes for approximate and exact inference, and a “stochastic memoizer” that allows for the description of non-parametric models.

ProbLog [92] is a probabilistic extension of the Prolog logical programming language. As in Prolog, a program in ProbLog consists of a set of facts and clauses. However, in ProbLog, each fact or clause is labeled with a real value that determines
the probability that the fact or clause is true—thus, a “pure Prolog” program is one in which all facts or clauses have a probability of one. Program evaluation in ProbLog consists on finding possible solutions for a clause and computing the success probabilities of those solutions.

The probabilistic programming language BLOG [93] is designed for describing probability models with unknown objects. Programs in BLOG specify generative processes that define a probability distribution over possible worlds with first-order model structures. Inference in BLOG is achieved by employing a general-purpose rejection sampling algorithm.

Domain-specific languages are probabilistic programming languages designed for particular forms of computation, and have existed since the development of the discrete-event simulation language Simula [94].

BUGS [16] is a language for Bayesian inference and learning for data on hierarchical graphical models. Models in BUGS describe a generative process, and are compiled and translated into graph structures that the BUGS runtime can use for performing inference by way of MCMC simulation algorithms like Gibbs sampling and Metropolis-Hastings.

Stan [95] is a probabilistic programming language and software library implementing Bayesian inference and maximum-likelihood estimation. Inference over generative processes described in the Stan language is performed using specialized the MCMC algorithms known as Hamiltonian Monte-Carlo and No-U-Turn sampling [96].

Libraries for probabilistic programming allow for specifying certain forms of non-deterministic computation in the context of a conventional, deterministic programming language. Factorie [97] is a library for probabilistic programming with so-called imperatively defined factor graphs written for the Scala language, with capabilities for
inference and MCMC simulation. Infer.NET [98] is a library for Bayesian inference in graphical models written for C#, with support for multiple inference algorithms.

2.2 Background

2.2.1 Sequential Hypothesis Testing

Statistical hypothesis tests are widely used for drawing inferences about a population given a finite number of observations. The objective is to use a test statistic to accurately decide between two competing hypotheses [99]: a null or “default” hypothesis $H_0$, and an alternative hypothesis $H_1$ that is true if $H_0$ is rejected.

The particular kind of hypothesis test relevant to the work presented in this thesis is as follows. Let $\pi$ denote the unknown probability that a given Boolean test predicate $\phi$ is true for the general population. We are given a set of $n$ observations, and the measurements $X_1, X_2, \ldots, X_n$ where $X_i = 1$ if $\phi$ is true for the $i$th observation, zero otherwise. For an arbitrary probability threshold $p$, we choose between the hypotheses:

$$H_0 : \pi \leq p \quad \text{or} \quad H_1 : \pi > p$$

To control the probability of Type-I (false positive) and Type-II (false negative) errors, the test is formulated as a Neyman-Pearson likelihood ratio test [100], which requires the use of simple hypotheses:

$$H_0 : \pi = p - \epsilon \quad \text{or} \quad H_1 : \pi = p + \epsilon$$

where $\epsilon$ is a small constant. Note that each $X_i$ is a Bernoulli trial, adding up to $k$ successful trials out of $n$. Therefore, the test decides between $H_0$ and $H_1$ by examining
the ratio between the likelihood functions

\[ L_0 = \text{Binomial}(k \mid n, p - \epsilon) \quad \text{and} \quad L_1 = \text{Binomial}(k \mid n, p + \epsilon) \]

The sample size \( n \) is determined by the constant values \( \alpha \) and \( \beta \), which bound the Type-I and Type-II error rates respectively, so that

\[ P(\text{choosing } H_1 \mid H_0 \text{ is true}) < \alpha \quad \text{and} \quad P(\text{choosing } H_0 \mid H_1 \text{ is true}) < \beta. \]

The Sequential Probability Ratio Test [101], or SPRT for short, is a Neyman-Pearson hypothesis test that does not require fixing \( n \) before drawing the sample. Instead, the SPRT takes a single observation at a time, leading to one of the following three outcomes:

1. Accept \( H_0 \) and reject \( H_1 \), or
2. Reject \( H_0 \) and accept \( H_1 \), or
3. Undecided. Need more observations.

If the result is “undecided,” a new observation is drawn, used to update the likelihood ratio of the test, and the process is repeated. The SPRT guarantees convergence to either accepting or rejecting \( H_0 \) after a finite amount of independent and identically distributed (i.i.d.) observations.

Consider the standard Binomial test defined above. First, the likelihood thresholds for the two hypotheses are defined as follows:

\[ \tau_0 = \log\left(\frac{\beta}{1 - \alpha}\right) \quad \text{and} \quad \tau_1 = \log\left(\frac{1 - \beta}{\alpha}\right). \]
Figure 2.1: Likelihood ratio (a) and proportion of successful trials (b) as a function of the number of values observed by a SPRT.

Let $n$ denote the number of observations that the SPRT has taken, with $k_n$ successful trials starting at $k_0 = 0$ so that, after the SPRT takes measurement $x_n \in \{0, 1\}$, we have $k_n = k_{n-1} + x_n$. Thus, the likelihood ratio of the test after the $n^{th}$ observation is

$$
\lambda_n = k_n \log \left( \frac{p + \epsilon}{p - \epsilon} \right) + (n - k_n) \log \left( \frac{1 - (p + \epsilon)}{1 - (p - \epsilon)} \right).
$$

Then, the value of $\lambda_n$ is used to determine if a decision can be made: If $\lambda \leq \tau_0$, the test accepts $H_0$ and rejects $H_1$. Conversely, if $\lambda \geq \tau_1$, the test rejects $H_0$ and accepts $H_1$. If none of these conditions is met, more observations are needed and therefore the test stays undecided.

The plots in Figure 2.1 show the progress of a SPRT with $\pi = 0.9$, $p = 0.95$, $\epsilon = 0.005$ and $\alpha = \beta = 0.01$. As values are observed, the proportion of successful trials $\frac{k_n}{\pi}$ approaches $\pi$ and the likelihood ratio $\lambda$ crosses the threshold $\tau_0$, which leads
it to accept $H_0$ and reject $H_1$.

The work presented in Chapter 3 employs a particular version of the SPRT known as the End-Biased Test [102] that is directed at environments (e.g., databases) where the difference between $\pi$ and the threshold $p$ is often large and a hypothesis can be accepted or rejected with ease.

### 2.2.2 Cost-based Query Optimization

As mentioned in Section 2.1, query optimization techniques in relational databases are generally classified as rule-based or cost-based, the latter consisting of an iterative search process that generates new plans and calculates their costs to find a plan of minimal cost [25,26]. In this context, the “cost” of a plan is an estimate of the amount of resources required to execute it.

To illustrate the process of query optimization, consider a simple database schema similar to the one introduced in Example 1.2, with an additional table `region`.*. The tables in this schema are:

- `region(region_id,name)` which contains 5 records. Each record is identified by a unique key value for `region_id`.
- `nation(nation_id,name,region_id)` which contains 25 records. Each record is identified by the key attribute `nation_id`. The attribute `region_id` is a foreign key that references a valid record from `region`.
- `customer(customer_id,name,balance,nation_id)` with 1,000 records. Each record is identified by the key attribute `customer_id`. The attribute `nation_id` is a foreign key that references a valid record from `nation`.

---

*Both of these schemata are based on the TPC-H benchmark’s data warehousing star schema. See [http://www.tpc.org/tpch/](http://www.tpc.org/tpch/)
Suppose we are interested in finding out the names of all European customers that have a negative account balance. In the database, Europe is recorded in the region to which some records in nation make reference via the region_id attribute. Therefore, the query finds the name of each customer with negative balance whose nation_id refers to a nation with a region_id matching that of Europe. This can be expressed with the SQL statement:

```sql
SELECT c.name
FROM customer AS c, nation AS n, region AS r
WHERE c.nation_id = n.nation_id
AND n.region_id = r.region_id
AND c.balance < 0
AND r.name = 'EUROPE';
```

Before a SQL query can be optimized and executed, it is translated into a plan that describes it as a sequence of operations that produce the desired result. Translating the above query into relational algebra is straightforward, resulting in the naïve query plan:

\[
\pi_{c.name}\left(\sigma_{c.nation_id=n.nation_id}\left(\sigma_{n.region_id=r.region_id}\left(\sigma_{c.balance<0}\left(\sigma_{r.name='EUROPE'}\left(r \times (c \times n)\right)\ldots\right)\right)\right)\right)
\]

where \(\pi\) is the projection operation, \(\sigma\) is the selection or filter operation, and \(\times\) is the Cartesian product operation. A complete description of the semantics of relational algebra can be found in [22].

Note that the above query plan is merely one of many possible expressions for the same query. Thus, query optimizers rely on the algebraic properties of relational operations to search for alternative formulations of a given plan. Examples of simple transformations based on such equivalences include:
• Re-arranging selections: $\sigma_{\phi_1}(\sigma_{\phi_2}(R)) \equiv \sigma_{\phi_2}(\sigma_{\phi_1}(R))$.

• Switching Cartesian products: $R \times S \equiv S \times R$, also $R \times (S \times T) \equiv (R \times S) \times T$.

• Constructing joins: $\sigma_{R.a=S.b}(R \times S) \equiv R \bowtie_{R.a=S.b} S$.

Applying these transformations to the naïve plan leads to the expression

$$\pi_c \text{name} \left(\sigma_{c \text{balance}<0} \left(\sigma_{r \text{name}='EUROPE'} \left(\begin{array}{l} R \bowtie_{n \text{region_id}=r \text{region_id}} (C \bowtie_{c \text{nation_id}=n \text{nation_id}}) \cdots \end{array}\right)\right)\right)$$

which can be seen represented as a tree in Figure 2.2.

One of the most commonly used transformations in query optimization consists of pushing selection operations below joins onto the lower levels of the query plan:

$$\sigma_{\phi[S,c]}(R \bowtie_{R.a=S.b} S) \equiv R \bowtie_{R.a=S.b} (\sigma_{\phi[S,c]}(S)).$$
The rationale for moving selections as low as possible is that doing so reduces the number of records that subsequent operations must take as input, thereby reducing their execution costs. Following our example, the equivalent plan $P_2$ shown in Figure 2.2 can be obtained after the selections on the attributes $c\text{.balance}$ and $r\text{.name}$ have been pushed below the join operations.

Join operations are always commutative, and are associative as long as the attributes referenced in the join predicate are present:

$$R \bowtie_{R.a=S.b} (S \bowtie_{S.c=T.d} T) \equiv (R \bowtie_{R.a=S.b} S) \bowtie_{S.c=T.d} T.$$ 

In our example, plan $P_3$ can be obtained by shuffling the order of the joins between customer, nation and region.

To demonstrate how a cost-based optimizer picks a query plan, consider a simple cost model in which the cost of a plan $P$ is the operational cost of the topmost operation plus the cost of its children sub-plans $P^{(1)}, P^{(2)}, \ldots, P^{(n)}$:

$$\text{cost}(P) = g[P] + \sum_{i=1}^{n} \text{cost}(P^{(i)})$$

where the operational cost function $g$ is defined for each operation type:

$$g[\sigma_{\phi}(R)] = |R|$$
$$g[\pi_{A_1,A_2,\ldots}(R)] = |R|$$
$$g[R \bowtie_{R.a=S.b} S] = |R| + |S|.$$
Then, the cost equations for the example plan $P_1$ expand as follows:

\[
\text{cost}(P_1) = g\left[\pi_{c.\text{name}}(P_1^{(1)})\right] + \text{cost}\left(P_1^{(1)}\right)
\]
\[
= g\left[\pi_{c.\text{name}}(P_1^{(1)})\right] + g\left[\sigma_{c.\text{balance}<0}(P_1^{(2)})\right] + \text{cost}\left(P_1^{(2)}\right)
\]
\[
\vdots
\]
\[
= g\left[\pi_{c.\text{name}}(P_1^{(1)})\right] + g\left[\sigma_{c.\text{balance}<0}(P_1^{(2)})\right] + g\left[\sigma_{r.\text{name}='EUROPE'}(P_1^{(3)})\right]
\]
\[
+ \cdots + g\left[c \bowtie_{c.\text{nation_id}=n.\text{nation_id}}\right]
\]

Note that this cost model requires also an estimate of the number of output records for each sub-plan. Assume that, in our database, 200 customers are located in a European nation, 50 customers have negative balance, and 5 nations are located in Europe.

Observe that the order of the joins in $P_1$ and $P_2$ is the same. However, the joins in $P_1$ have to process the three tables in their entirety, while the joins in $P_2$ have to process smaller, filtered versions. Following our cost model, the costs of the joins in $P_1$ would be

\[
g\left[c \bowtie_{c.\text{nation_id}=n.\text{nation_id}}\right] = 1,000 + 25 = 1,025
\]
\[
g\left[n \bowtie_{n.\text{region_id}=r.\text{region_id}}\right] = 1,000 + 5 = 1,005
\]

while the costs of the join operations in $P_2$ are

\[
g\left[c \bowtie_{c.\text{nation_id}=n.\text{nation_id}}\right] = 200 + 25 = 225
\]
\[
g\left[n \bowtie_{n.\text{region_id}=r.\text{region_id}}\right] = 200 + 1 = 201
Therefore, $P_2$ is preferable to $P_1$.

Regarding $P_3$, notice that the selections remain pushed to the lower levels of the plan, but the structure of the join operations is different. The join between `nation` and `region` is inexpensive and produces five records, thus reducing the cost of the join with `customer`. The operational costs for these operations are

\[
g\left[\n \text{nation} \Join \text{region} \mid n.\text{region}_id = r.\text{region}_id \right] = 25 + 5 = 30
\]
\[
g\left[\n \text{customer} \Join \text{nation} \mid c.\text{nation}_id = n.\text{nation}_id \right] = 200 + 5 = 205
\]

Therefore, $P_3$ is preferable to $P_2$. Finally, the optimizer chooses $P_3$ as the plan of minimal cost.

### 2.2.3 Gibbs Sampling for Generative Models

Generative processes are popular in Bayesian statistics and machine learning, particularly in models containing “hidden” or latent variables that are not directly observed from the data [103]. A generative process models the data set as a sequence of steps that, given some parameters, produces the observable data by way of sampling.

To illustrate this, let us consider an example generative model for simple linear regression. In linear regression, the goal is to infer the best fitting values $\alpha$ and $\beta$ in the equation

\[
y = \alpha x + \beta
\]

given observed values $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$. To model this problem as a generative process, one posits a distribution for the observed data and, optionally, other parameters (here, “$\sim$” denotes “is sampled from”):
\[
\begin{align*}
\alpha & \sim \text{Normal}(0, \sigma_0^2) \\
\beta & \sim \text{Normal}(0, \sigma_0^2) \\
\sigma^2 & \sim \text{InverseGamma}(1, 1) \\
\text{for each } i \in \{1, 2, \ldots, n\}: \\
y_i & \sim \text{Normal}(\alpha x_i + \beta, \sigma^2)
\end{align*}
\]

The model assumes that each data point \( y_i \) is normally distributed with variance \( \sigma^2 \) and mean \( \alpha x_i + \beta \). These variables, in turn, have their own distributions that act as “priors” to the observable data, forming a hierarchy of conditional dependence relationships that starts at the constant hyper-parameter \( \sigma_0^2 \) and can be represented as a graph [104]. Figure 2.3 shows the graphical, “plate notation” representation of the model, where the edges represent conditional dependence relations between the model variables, random variables are represented with circular nodes, constants correspond to square nodes and the gray background on a node labels the variable as observable. This model corresponds to the joint probability density function (PDF)
of the parameters and the observable data:

\[
p(\alpha, \beta, \sigma^2, \{y_i\}|\{x_i\}, \sigma^2_0) = \text{Normal}(\alpha|0, \sigma^2_0) \times \text{Normal}(\beta|0, \sigma^2_0) \times \text{InverseGamma}(\sigma^2|1, 1) \times \prod_{i=1}^{n} \text{Normal}(y_i|\alpha x_i + \beta, \sigma^2)
\]  

(2.1)

Given that \(\{y_i\}\) has been observed, the objective is to infer the parameters \(\alpha, \beta\) and \(\sigma^2_0\) by analyzing the conditional, “posterior” distribution

\[
P(\alpha, \beta, \sigma^2|\{x_i\}, \{y_i\}, \sigma^2_0).
\]

In practice, these functions are rarely amenable to closed-form analysis, and sampling techniques have to be employed. One of the most widely used algorithms for this class of models is Gibbs sampling [11, 105], an iterative MCMC algorithm for drawing samples from multivariate distributions. Given the set of random variables \(Z = \{Z_1, Z_2, \ldots, Z_m\}\), a Gibbs sampler draws values from the distribution \(P(Z)\) by separately drawing from each of the \(m\) conditional distributions \(P(\{Z_i\}|Z \setminus \{Z_i\})\) in this fashion:

\[
\begin{align*}
z^{[j]}_1 & \sim P\left(Z_1|z^{[j-1]}_2, z^{[j-1]}_3, \ldots, z^{[j-1]}_m\right) \\
z^{[j]}_2 & \sim P\left(Z_2|z^{[j]}_1, z^{[j-1]}_3, \ldots, z^{[j-1]}_m\right) \\
& \vdots \\
z^{[j]}_{m-1} & \sim P\left(Z_{m-1}|z^{[j]}_1, z^{[j]}_2, \ldots, z^{[j]}_{m-2}, z^{[j-1]}_m\right) \\
z^{[j]}_m & \sim P\left(Z_m|z^{[j]}_1, z^{[j]}_2, \ldots, z^{[j]}_{m-1}\right)
\end{align*}
\]

where \(z^{[j]}_i\) denotes the value of variable \(Z_i\) in the \(j^{th}\) generation of samples for \(P(Z)\).

In the linear regression example, a Gibbs sampler can be derived by applying
Bayes’ rule on Equation 2.1, resulting in the PDFs

\[
p\left(\alpha \mid \beta, \sigma^2, \{x_i\}, \{y_i\}, \sigma_0^2\right) = \text{Normal} \left( \frac{\sum_{i=1}^{n} x_i(y_i - \beta)}{\sigma^2 + \sum_{i=1}^{n} x_i^2}, \frac{\sigma^2 \sigma_0^2}{\sigma^2 + \sigma_0^2 \sum_{i=1}^{n} x_i^2} \right)
\]

\[
p\left(\beta \mid \alpha, \sigma^2, \{x_i\}, \{y_i\}, \sigma_0^2\right) = \text{Normal} \left( \frac{\sum_{i=1}^{n} y_i - \alpha x_i}{\sigma^2 + n \sigma_0^2}, \frac{\sigma^2 \sigma_0^2}{\sigma^2 + n \sigma_0^2} \right)
\]

\[
p\left(\sigma^2 \mid \alpha, \beta^2, \{x_i\}, \{y_i\}, \sigma_0^2\right) = \text{InverseGamma} \left( \sigma^2 \left| 1 + \frac{n}{2}, 1 + \frac{\sum_{i=1}^{n} (y_i - \alpha x_i - \beta)^2}{2} \right. \right)
\]

Notice that each of the above functions are conditioned on each other. In a Gibbs sampler, a non-circular order needs to be determined for sampling the variables during every generation, by conditioning some of the variables on the previous generation. For example, the following order is valid for generation \(i\):

1. Sample \(\beta[i]\) conditioned on \(\alpha[i-1]\) and \(\sigma^2[i-1]\),
2. Sample \(\sigma^2[i]\) conditioned on \(\alpha[i-1]\) and \(\beta[i]\),
3. Sample \(\alpha[i]\) conditioned on \(\sigma^2[i]\) and \(\beta[i]\).

This order implies that, to sample generation \(i = 1\), some “zeroth” generation of start-up values must be set. A possible way to do this is to set this generation as follows:

1. Sample \(\beta[0] \sim \text{Normal}(0, \sigma_0^2)\),
2. Sample \(\sigma^2[0] \sim \text{InverseGamma}(1, 1)\),
3. Sample \(\alpha[0]\) conditioned on \(\sigma^2[0]\) and \(\beta[0]\).
Figure 2.4 shows the values of parameters $\alpha[i]$ and $\beta[i]$ for the first 30 iterations of the Gibbs sampler, as each converge to the true values $\alpha \approx -9.75$ and $\beta \approx 6.65$. The period starting at iteration $i = 0$ until all the variables in a Gibbs sampler converge is known as its “burn-in period.”

2.3 Overview of MCDB/SimSQL

The Monte Carlo Database, or MCDB for short [10], is an RDBMS that allows for the presence of uncertain data and provides the means for processing queries on such data. To this end, MCDB permits the creation of database tables containing stochastic attributes—that is, attributes that are associated with finite distributions of possible values, alongside conventional, deterministic attributes.

MCDB aims at providing a general framework that can accommodate a large class
of user-defined uncertainty models. To this end, MCDB follows a pure sampling-based approach, drawing random values from user-defined library functions, known as Variable Generator (VG) functions, that encode the uncertainty model and can be parameterized with the contents of the database or the results of a query. The process of query evaluation in MCDB is shown in Figure 2.5.

To illustrate how stochastic analysis is carried out in MCDB, consider Example 1.1, wherein the goal is to determine the hypothetical profits of a retailer if the price of each item sold during the previous had been 5% higher. An analyst proposes a model\(^1\) for the demand curve of each item stored in the database, conceiving it as a simple linear curve between \(D_0\), which denotes the demand at a price of zero, and \(P_0\), which denotes the price when a quantity of zero items is purchased. Both \(D_0\) and

\(^1\) Taken from Jampani, et al [12].
Figure 2.6: Sampling linear demand curves from a stochastic model.

$P_0$ are modeled as being generated from Gamma distributions:

\[ D_0 \sim \text{Gamma}(k_d, \theta_d) \]
\[ P_0 \sim \text{Gamma}(k_p, \theta_p) \]

where $k_d$ and $k_p$ are the “shape” parameters, and $\theta_d$ and $\theta_p$ are the “scale” parameters. These parameters are estimated from the contents of a data warehouse that records the detail of every past order—that is, the quantity and price of each item in the purchase. The model, then, estimates the hypothetical quantity of each item in the order details from the previous year given a sale price that is 5% higher. As demonstrated in Figure 2.6, this is achieved by using the stored quantity $d$ and price $p$ from each item in an order to sample a pair $(D_0, P_0)$ conditioned so that the demand
curve passes through the point \((d, p)\):

\[
(D_0, P_0) \sim \left( \text{Gamma}(k_d, \theta_d), \text{Gamma}(k_p, \theta_p) \right| D_0 \left(1 - \frac{p}{P_0}\right) = d ) .
\]

Once a value for \((D_0, P_0)\) has been sampled from this distribution, the hypothetical quantity \(D_{\hat{p}}\) is calculated as

\[
D_{\hat{p}} = D_0 \left(1 - \frac{\hat{p}}{P_0}\right)
\]

where \(\hat{p} = 1.05p\) is the alternative price. After generating the hypothetical quantities for each sale from the previous year, the hypothetical revenue is computed. This whole process of sampling and aggregation is repeated \(N\) times to obtain an empirical distribution of the hypothetical revenue with \(N\) values.

Let us describe how this analysis is done in MCDB. The information on past orders is stored in the table `order_detail(order_id, item_id, price, qty)` where each record contains the sale price and purchased quantity for a given item in a particular order and the table `params(item_id, d0_shape, d0_scale, p0_shape, p0_scale)` containing the estimated parameters \(k_d, \theta_d, k_p, \theta_p\) for each particular item. The stochastic model is implemented as the VG function `BayesDemand` as described in [12]. Then, the stochastic table `order_est` containing the hypothetical quantities is created with the following SQL statement:
CREATE TABLE order_est(order_id, item_id, price, hyp_qty) AS
FOR EACH od IN order_detail
    WITH de AS BayesDemand(
        SELECT od.qty, od.price,
            pa.d0_shape, pa.d0_scale,
            pa.p0_shape, pa.p0_scale
        FROM params AS pa
        WHERE pa.item_id = od.item_id
    )
    SELECT od.order_id, od.item_id, od.price*1.05,
        de.value_d0*(1-((od.price * 1.05)/de.value_p0)
    FROM de;

To generate order_est, MCDB loops through each record od in order_detail, obtains the shape and scale parameter values corresponding to the item referenced by od.item_id in the params table, and uses these values to parameterize the BayesDemand VG function. For each od, a random sample of $N$ values is drawn from the VG function and bound to the temporary table de, which contains the generated values for $D_0$ and $P_0$ as the stochastic attributes value_d0 and value_p0, from which the value of hyp_qty is calculated.

After order_est has been generated, it can be referenced and used to answer SQL queries, in conjunction with other deterministic and stochastic tables present in the database. If a user wishes to calculate the total hypothetical revenue, the following SQL query suffices:

```sql
SELECT SUM(oe.price * or.hyp_qt) AS revenue
FROM order_est AS oe;
```

Figure 2.7 shows an example of a generated order_est and the output for the above aggregation query. Note how the stochastic attributes hyp_qty and revenue contain a constant number of multiple values in all records in the table. This array-based representation is a “tuple bundle,” which contains the attribute values for
<table>
<thead>
<tr>
<th>order_id</th>
<th>item_id</th>
<th>price</th>
<th>hyp_qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BOLT</td>
<td>1.05</td>
<td>20 25 16</td>
</tr>
<tr>
<td>1</td>
<td>NUT</td>
<td>2.10</td>
<td>10 5 6</td>
</tr>
<tr>
<td>2</td>
<td>WASHER</td>
<td>5.25</td>
<td>4 4 2</td>
</tr>
<tr>
<td>2</td>
<td>NUT</td>
<td>2.10</td>
<td>20 15 11</td>
</tr>
<tr>
<td>3</td>
<td>WASHER</td>
<td>5.25</td>
<td>12 8 6</td>
</tr>
<tr>
<td>4</td>
<td>BOLT</td>
<td>1.05</td>
<td>40 55 50</td>
</tr>
</tbody>
</table>

Generated order_est table.

<table>
<thead>
<tr>
<th>revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>210.00</td>
</tr>
<tr>
<td>189.00</td>
</tr>
<tr>
<td>147.00</td>
</tr>
</tbody>
</table>

Result of aggregate query over order_est.

Figure 2.7: Example output query in MCDB, containing a stochastic attribute with $N = 3$ possible worlds.

various possible worlds [62,67] or instances of the database that result from random sampling, each column in the attribute representing a separate instance. Thus, the stochastic attribute revenue is obtained by aggregating each column of hyp_qty separately, multiplying each value by the corresponding price from the same row.

The remainder of this section describes the tuple bundle format for stochastic tables, the interface through which VG functions interact with the rest of the system, the basics of the MCDB query processing for relational algebra queries, and the SimSQL extensions for machine learning.
2.3.1 The MCDB Data Model

In MCDB, a database table is a set of tuple bundles that follow the same schema \( S \). In general, a tuple bundle \( t \) is an array of \( N \) records with schema \( S \), designated as \( t[1], t[2], \ldots, t[N] \), where \( t[i] \) denotes the value of \( t \) in the \( i^{th} \) possible database instance. The constant \( N \) is a system-wide parameter that determines the number of possible instances created during simulation and sampling, so that all the tuple bundles being processed by a given query have exactly \( N \) records.

Given a tuple bundle \( t \) with schema \( (\text{att}_1, \text{att}_2, \ldots, \text{att}_K) \), the value of the attribute \( t.\text{att}_j \) in the \( i^{th} \) possible instance is denoted as \( t[i].\text{att}_j \). Besides having a particular data type, attributes can be either deterministic or stochastic—if \( t[i].\text{att}_j \) has the same value in all \( t[i] \) regardless of \( N \), then \( t.\text{att}_j \) is said to be deterministic; otherwise, it is stochastic. Furthermore, each tuple bundle is associated with a special Boolean attribute called \( t.\text{isPresent} \) that determines if a given record exists in a particular instance, so that \( t[i].\text{isPresent} = \top \) if \( t \) exists in the \( i^{th} \) possible world, \( \bot \) otherwise.

Tuple bundles allow MCDB to generalize the operations in a relational database to process multiple possible database instances simultaneously, and provide a compact and data-local record representation. All of the \( N \) possible values associated with a stochastic attribute are stored together in an array, and deterministic attributes are stored as singletons. This applies to \( \text{isPresent} \) as well, with the provision that, if \( t[i].\text{isPresent} = \bot \) for all \( t[i] \), then \( t \) does not exist in any possible instance and does not need to be stored. Note that, if \( \text{isPresent} \) is not stored in \( t \), then it is assumed that \( t[i].\text{isPresent} = \top \) for all \( t[i] \).

For example, consider the following scenario: a user wishes to store a table with schema \( (\text{client, balance}) \), where \( \text{customer} \) is a deterministic string and \( \text{balance} \) is a stochastic integer. Assume \( N = 3 \). Information for clients ACME,
<table>
<thead>
<tr>
<th>customer</th>
<th>balance</th>
<th>.isPresent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACME</td>
<td>25 -10</td>
<td>1</td>
</tr>
<tr>
<td>ADVENT</td>
<td>20 1</td>
<td>-1</td>
</tr>
<tr>
<td>FOO</td>
<td>-10 -20</td>
<td>-25</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>customer</th>
<th>balance</th>
<th>.Present</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACME</td>
<td>25 -10</td>
<td>T ⊥ T</td>
</tr>
<tr>
<td>ADVENT</td>
<td>20 1</td>
<td>T ⊥ ⊥</td>
</tr>
</tbody>
</table>

(b)

Figure 2.8: Compact representation of tuple bundles with $N = 3$ possible database instances, before and after the application of a selection operation.

ADVENT and FOO is stored, the latter having negative account balances according to all the possible database instances. These tuple bundles would have the form described in Figure 2.8(a) which, after being processed by a selection operation that discards records with negative account balance, obtain explicit `isPresent` attributes as shown in Figure 2.8(b).

### 2.3.2 VG Functions and Stochastic Table Creation

Variable Generator (VG) functions are the main source of uncertainty in MCDB and supply the values for stochastic attributes via random sampling. The relational operation `VGWrapper` is responsible for loading VG functions, parameterizing them and obtaining their organizing their output samples as tuple bundles.

VG functions are user-defined external libraries that follow a specific interface and generate sample values for a single possible world or database instance at a time. The methods that a VG function class makes publicly available are:

1. `ClearParams`: notifies the VG function that it is about to be parameterized.

   At this point, the function can discard the parameters from previous invocations.
2. **TakeParams**: takes an input record \( t_{in} \) and updates the set of parameters with its contents. This method can be called multiple times, since some parameters may span multiple records or come from different tables.

3. **TakeSeed**: takes an input value \( s \), which must be used to seed the random number generator that will be used to draw samples.

4. **OutputVals**: returns a record \( t_{out} \) containing the sampled attribute values for a single possible world or database instance. These samples can span multiple records. To signal that all the samples for the current possible world have been generated, the method returns \( t_{out} = \text{NULL} \).

VG function invocations are always present in the special SQL `CREATE TABLE` statements for defining stochastic tables. To translate this statement from SQL into a relational algebra expression, MCDB relies on the `Seed` and `VGWrapper` operations.

The `Seed` operation takes an input table \( T \) and appends to every tuple \( t \in T \) a new deterministic attribute \( s \) (inaccessible to the end-user) with a distinct, randomly-generated value. This attribute serves various purposes, such as identifying tuple bundles belonging to the same group, seeding the random number generator used by the VG function, and serving as a compressed representation of the stochastic attributes produced by said function.

The `VGWrapper` operation takes an “outer” input table \( T_O \) containing seed attribute \( s \), a set of “inner” input tables (which may contain \( s \)) \( T_1, T_2, \ldots, T_k \), and a VG function \( f \). For each outer tuple \( t \in T_O \) and each possible world \( i = 1, 2, \ldots, N \), `VGWrapper` invokes \( f \) with parameters \( t[i] \) and all the inner tuples \( t'[i] \in T_j, j \in \{1, \ldots, k\} \) with matching seed attribute values \( t.s = t'.s \) (or the entire table if \( T_j \) does not contain \( s \)). After finishing all the invocations for \( t \), the new tuple bundles are
assembled, containing the new stochastic attributes and the original attributes from $t$.

As mentioned previously, instantiating stochastic tables requires a special SQL CREATE TABLE statement that includes VG function invocation and parameterization. The general structure of this statement is as follows:

```
CREATE TABLE \{tableName\} ((att_1), (att_2), ..., (att_k)) AS
  FOR EACH \{outerTupleAlias\} IN \{outerQuery\}
    WITH \{vgTableAlias\} AS \{vgFunctionName\} ( 
      \{innerQuery_1\},
      \{innerQuery_2\},
      \vdots,
      \{innerQuery_k\}
    )
  \{innerQuery\}
```

The above statement is translated into a sequence of relational operation that prepare the input to the VGWrapper operation and collect its output. As displayed in Figure 2.9(a), this sequence contains:

1. The outerQuery is executed and its output $T'_O$ is used to generate the outer table $T_O = \text{Seed}(T'_O)$.

2. Each innerQuery is executed and its output $T'_j$ joined with $T_O$ to produce the inner table $T_j$. If the inner query already has a join operation that involves an attribute from $T_O$ (referenced via outerTupleAlias), then $T_j = T'_j$. Otherwise, the tables are joined with a Cartesian product: $T_j = T_O \times T'_j$.

3. The VGWrapper operation is executed to produce the temporary table $T^*$, taking as input the outer table $T_O$ and all the inner tables $T_1, T_2, ..., T_k$. The VG function referred by vgFunctionName is used to generate samples for the $N$ possible instances.
Figure 2.9: Query plans for instantiating MCDB stochastic tables.

4. The finalQuery is executed to create the table tableName. The attribute values $att_1, att_2, \ldots, att_m$ can be retrieved from $T^*$ (referenced via vgTableAlias), or from $T_O$ (referenced via outerTupleAlias), or from any additional tables specified in the FROM clause.

2.3.3 Query Evaluation

To execute standard SQL queries that involve stochastic tables, MCDB generalizes and extends the standard relational operation to handle tuple bundles appropriately, preserving the semantics of the relational model.
Some of the standard relational operations, such as joins and aggregates, rely on the ability to order and group records by a specific attribute value. However, this is a problematic task when dealing with stochastic attributes that contain multiple values. To manage this, MCDB introduces a third special purpose operation called Split that can be applied on the input set of another operation. Essentially, Split takes a tuple bundle $t$ and “transfers” the uncertainty of a given stochastic attribute $t.a$ into the $t.isPresent$ attribute, effectively making $t.a$ deterministic and suitable for ordering and grouping. To achieve this, Split enumerates all the $k \leq N$ distinct values of $t[i].a$ for which $t[i].isPresent = \top$ and proceeds to create $k$ different copies of $t$; each copy $t'$ contains a single, different value $t'.a$, and $t'[i].isPresent = \top$ for all $i$ where $t[i].a = t'.a$. For example, consider a tuple bundle with schema $(name, age, isPresent)$ where age is stochastic and $N = 4$:

| Betty | 26 | 25 | 28 | 25 | T | T | ⊥ | T |

Applying Split on the attribute age produces two separate tuple bundles:

| Betty | 26 | T | ⊥ | ⊥ | ⊥ |
| Betty | 26 | ⊥ | T | ⊥ | T |

Notice that no tuple bundles for age=28 exist, since the possible instance containing this value was marked as invalid in the isPresent attribute.

In MCDB, the standard relational operations rely heavily on the isPresent attribute to perform computations on multiple database instances at once and represent their results. For example, the selection operation takes a tuple bundle $t$, applies the
Boolean predicate $\phi$ on each separate instance $t[i]$, updating the $isPresent$ attribute:

$$t[i].isPresent = t[i].isPresent \land \phi(t[i])$$

If the above results in $isPresent[i] = \bot$ for all $i$, then $t$ does not become part of the output set, as shown in Figure 2.8.

Another example of the importance of $isPresent$ is the aggregation operation. Suppose a user wants to compute the $SUM$ aggregate of the $balance$ attribute from Figure 2.8(b) and that the result is stored in a single tuple bundle $t'$ with stochastic attribute $sumVal$. MCDB uses $isPresent$ to determine which attribute values should be counted for computing the aggregate, having

$$t'[i].sumVal = \sum_{i} t[i].acctBal \times I(t[i].isPresent)$$

where $I(x)$ is an indicator function that returns 1 if $x = \top$, zero otherwise. Then, the resulting $isPresent$ attribute is also “aggregated”

$$t'[i].isPresent = \bigvee_{i} t[i].isPresent$$

Leading to the result tuple bundle:

```
| 25 | 1 | 1 | T | T | T |
```

Notice that the above aggregation did not require the application of the $Split$ operation on the input table. This is because, although stochastic attributes were involved in the aggregate computation, none of them were used for grouping. Similarly, a join operation does not require the application of $Split$ if the attributes of the joining
predicates are all deterministic.

2.3.4 MCMC Simulation with SimSQL

SimSQL [13] is an extension of MCDB that aims to provide support for statistical inference on a wide range of generative models in a distributed, “Big Data” environment. SimSQL allows for the specification and execution of a Gibbs sampler such as the above, by making use of stochastic tables that are updated in mutually recursive fashion.

A Gibbs sampler can be specified in SimSQL via multiple `CREATE TABLE` statements, with one for each random variable in the sampler. Consider the linear regression example from Section 2.2.3 †, wherein conditional distributions for the parameters $\alpha$, $\beta$ and $\sigma^2$ were described. Since these distributions are defined in terms of each other, it would be impossible to specify them using the conventional `CREATE TABLE` statements from MCDB. To avoid such circular definitions, stochastic tables in SimSQL are associated with a generation index, allowing for a mutually recursive specification of the Gibbs sampler.

To specify the tables in a sampler, the user must first decide a non-circular order in which the tables must be processed, based on the formal definition of the Gibbs sampler and making use of the aforementioned generation indices. Following the order proposed for the linear regression example, the SimSQL tables for the Gibbs sampler are defined accordingly:

1. $beta[i]$ referencing tables $alpha[i - 1]$ and $sigma[i - 1]$,

2. $sigma[i]$ referencing tables $alpha[i - 1]$ and $beta[i]$, and

†More complex, real-word examples can be found in Appendix A.
3. \( \alpha[i] \) referencing tables \( \sigma[i] \) and \( \beta[i] \).

For a specification to be complete, the user must also supply `CREATE TABLE` statements for the “zeroth generation” to provide the variable initializations necessary for running generation \( i = 1 \). Which tables need to be defined depends on the order that the user has chosen—in the above example, \( \beta[0] \) and \( \sigma[0] \) have to be specified so that \( \alpha[i = 1] \) can be generated.

The following SQL statements provide the initializations for

\[
\beta \sim \text{Normal}\left(0, \sigma_0^2\right) \quad \text{and} \quad \sigma^2 \sim \text{InverseGamma}(1, 1)
\]

assuming that the value for the hyper-parameter \( \sigma_0^2 \) is stored as the attribute `hyperVar` in the single-record table `prior`.

```sql
CREATE TABLE beta[0](betaVal) AS
  WITH nb AS Normal(
    SELECT VALUES(0), p.hyperVar
    FROM prior AS p
  )
  SELECT nb.value
  FROM nb;
--

CREATE TABLE sigma[0](sigmaVal) AS
  WITH is AS InverseGamma(
    SELECT VALUES(1, 1)
  )
  SELECT is.value
  FROM is;
```

Assume that the data values \( x_1, x_2, \ldots, x_n \) and \( y_1, y_2, \ldots, y_n \) are stored as pairs in the database table `data(x,y)`. The SQL code to update the parameter \( \alpha \) is:

\[
\alpha \sim \text{Normal}\left( \frac{\sum_{i=1}^{n} x_i(y_i - \beta)}{\frac{\sigma^2}{\sigma_0^2} + \sum_{i=1}^{n} x_i^2}, \frac{\sigma^2 \sigma_0^2}{\sigma^2 + \sigma_0^2 \sum_{i=1}^{n} x_i^2} \right)
\]
for all iterations, including \( i = 0 \), would be:

```sql
CREATE TABLE alpha[i](alphaVal) AS
WITH na AS Normal(
  SELECT
    ( t.sumDiff / ((s.sigmaVal / p.hyperVar) + t.sumSq) ) AS meanV,
    (s.sigmaVal * p.hyperVar) / (s.sigmaVal + p.hyperVar * t.sumSq) ) AS varV
FROM sigma[i] AS s, prior AS p,
  (SELECT
    SUM(d.x * d.x) AS sumSq,
    SUM(d.x * (d.y - b.betaVal)) AS sumDiff
  FROM data AS d, beta[i] b) AS t)
SELECT na.value
FROM na;
```

In the above SQL statement, the temporary inner table \( t \) is used to compute two attributes: \( \text{sumSq} \), from the sum of squares \( \sum_{i=1}^{n} x_i^2 \); and \( \text{sumDiff} \), from the sum \( \sum_{i=1}^{n} x_i(y_i - \beta) \). Notice the references to the tables \( \text{beta}[i] \) and \( \text{sigma}[i] \), following the sampling order previously described.

Similarly, the code to update the parameter

\[
\beta \sim \text{Normal} \left( \frac{\sum_{i=1}^{n} y_i - \alpha x_i}{\sigma^2 / \sigma_0^2 + n}, \frac{\sigma^2 \sigma_0^2}{\sigma^2 + n \sigma_0^2} \right)
\]

for all iterations, starting with \( i = 1 \), would be:
CREATE TABLE beta[i](betaVal) AS
WITH nb AS Normal(
  SELECT (t.sumDiff / ((s.sigmaVal / p.hyperVar) + t.numData)) AS mean,
  (s.sigmaVal * p.hyperVar) / (s.sigmaVal + t.numData * p.hyperVar) AS var
FROM sigma[i-1] AS s, prior AS p,
  (SELECT COUNT(*) AS numData, SUM(d.y - a.alphaVal * d.x) AS sumDiff
   FROM data d, alpha[i-1] a) AS t)
SELECT nb.value
FROM nb;

As in the code for updating $\alpha$, the temporary inner table called $t$ is defined. However, this table contains the total number of data points $n$, represented as numData; and the sum $\sum_{i=1}^{n} y_i - \alpha x_i$, represented as sumDiff. Likewise, the references to the tables alpha[i-1] and sigma[i-1] follow the order specified in the Gibbs sampler.

Finally, the code for updating the parameter

$$
\sigma^2 \sim \text{InverseGamma}\left(1 + \frac{n}{2}, 1 + \frac{\sum_{i=1}^{n} (y_i - \alpha x_i - \beta)^2}{2}\right)
$$

for all iterations, starting with $i = 1$, is straightforward:

CREATE TABLE sigma[i](sigmaVal) AS
WITH is AS InverseGamma(
  SELECT 1 + COUNT(*)/2 AS shape,
  1 + 0.5 * SUM(
    pow(d.y - a.alphaVal * d.x - b.betaVal, 2)
  ) AS scale
FROM alpha[i-1] AS a, beta[i] AS b, data AS d)
SELECT is.value
FROM is;

With this SQL specification, SimSQL is able to construct a Gibbs sampler made of relational operations and VG functions to continuously update, iteration after
iteration, the values of the random variables.

Before executing a query, SimSQL must translate and optimize the set of `CREATE TABLE` statements that comprise the Gibbs sampler. Each individual statement is translated in the fashion described in Section 2.3.2. However, these mutually recursive statements also define the order in which the tables must be created for an iteration of the Gibbs sampler to be completed. It is important to note that, in SimSQL, the number of iterations for the Gibbs sampler is not related to the number of possible database instances or worlds, which defines the number of separate, parallel “chains.”

The directed acyclic graph (DAG) shown in Figure 2.10 corresponds to the order of stochastic table updates for the example linear regression model, up to iteration $i = 2$. To run this Gibbs sampler, SimSQL follows a frame-based approach, diving the dependency graph into a sequence of query plans, each plan roughly corresponding to the set of stochastic tables that comprise an entire iteration of the sampler. Dividing the chain into frames serves two purposes, namely:

1. To allow for check-pointing on the chain, so that the simulation may be stopped at any given time and resumed by the user later. To resume the simulation, it is necessary to materialize only the set of stochastic tables produced by the last
Figure 2.11: Two possible frame cuts for the linear regression Gibbs sampler, iterations $i = 0, 1$.

1. Successfully executed frame.

2. To assist the query optimizer by giving it reasonably small query plans with fresh statistics obtained from the last successfully executed frame. Given that cost-based optimizers rely on approximations of the running time and output set size of each operation, plans with fewer operations are “easier” to optimize and transform into fast, efficient plans.

Note that, for a given chain, there can be many possible frame cuts that satisfy these two conditions. SimSQL finds the cut that minimizes the size of the tables that are materialized after each plan is executed by creating a weighted dependency graph containing the nodes corresponding to the iterations $i - 1$ and $i$, treating this process as a max-flow/min-cut problem [106].

Figure 2.11 shows two possible frame cuts for the example dependency graph, for iterations $i = 0, 1$. Cut (a) leads to a plan that materializes $\beta[0], \sigma[0]$ and $\alpha[0]$, followed by a plan that materializes $\beta[1], \sigma[1]$ and
alpha[1]. On the other hand, cut (b) leads to a plan that materializes beta[0] and sigma[0], followed by a plan that materializes beta[1], sigma[1] and alpha[1]. In this case, cut (b) is preferable since it does not need to materialize alpha[0] after the plan has been executed.
Chapter 3

Evaluation of Probability Threshold Queries

As originally described, MCDB/SimSQL simply constructs an empirical distribution of result sets for a given query and returns it to the user for subsequent analysis. Thus, if an end-user wishes to draw inferences about the model, it is necessary to apply appropriate statistical analyses on the result sets returned by MCDB/SimSQL as a separate, external process.

Separating such analysis from the query evaluation engine has several drawbacks. First, it requires an end-user to apply an appropriate statistical analysis. Even for simple questions, this can be a complicated task that involves choosing and correctly employing an appropriate statistical hypothesis test to estimate the accuracy of the analysis.

Moreover, separating the analysis from the generation is likely to result in wasted system resources. Whenever a statistical analysis is performed, a crucial question is how many data is required; in the case of MCDB/SimSQL, additional observations must be generated via additional (possibly expensive) Monte Carlo iterations. If too many data is generated for a given analysis, it means resources have been wasted generating Monte Carlo iterations that are not needed; on the other hand, too little data means that the analysis cannot be performed with the desired accuracy. If the analysis is separated from the engine, then it is difficult to generate precisely the correct amount of data for the required analysis.

Pushing the analysis into MCDB/SimSQL allows the engine to determine exactly
the necessary amount of data to generate at different levels of granularity. For in-
stance, consider the what-if analysis from Example 1.1. The user wishes to find out
which items would have never been ordered after the hypothetical price increase, with
a probability of at least 95%. Imagine that MCDB/SimSQL runs a small number of
Monte Carlo iterations over the hypothetical demand for item X, and in every trial
the number of purchased units of X remained high, so it is immediately apparent that
there is little chance that item X will see a demand of zero units. Then, item X can
be rejected immediately.

In the remainder of this chapter, the problem of evaluating threshold queries in
MCDB/SimSQL is studied. These queries search for records (or groups of records)
that do or do not satisfy some Boolean predicate with a given, user-specified proba-
bility. Examples are: “Which regions will see more than a 2% decline in sales with
probability higher than 50%?” “Which packages will arrive late with probability of
at least 5%?” “What items led to an increase in revenue after a hypothetical price
change, with probability of 80% or higher?”

3.1 Overview of Threshold Queries

In Section 2.3, the table order_est(order_id,item_id,price,hyp_qty) was
defined as having the stochastic attribute hyp_qty generated by sampling from the
BayesDemand VG function, which encodes the stochastic model for the analysis
described in Example 1.1. Suppose that a user is interested in finding out, with 95%
probability, the items that customers would drop from at least one of their orders as
a result of the price increase. The syntax for the query is:
Given this statement, the MCDB/SimSQL query processing engine will first evaluate the query under the USING clause to instantiate the order_est table, generating samples for the hyp_qty attribute. Then, the records are partitioned using the item_id attribute, and a statistical test is applied to determine those items that appeared with quantity zero in some order, with high probability (> 0.95). The records in the result set contain the attributes item_id and res; the latter indicates the result of the hypothesis test for the item as either ACCEPT or REJECT.

The body of the TEST WHETHER clause defines the Boolean predicate of the hypothesis to be tested on each partition. The predicate must belong to one of the following three categories: EXISTS predicates, NOT EXISTS predicates, or aggregation predicates.

EXISTS predicates are used to test, on each partition, if there is at least one record that satisfies the specified condition. For instance, assume that the table orders(order_id, customer_id) stores the ID of the customer that placed each order in order_est, and the user wants to find out who are the customers that would have dropped an item from at least one of their orders. This can be done with the following query:
Observe that, in the above query, the USING clause contains an entire sub-query and its attributes are available to the rest of the threshold query statement. In general, this is always the case. Thus, a USING clause that contains only the name of a table is equivalent to the sub-query SELECT * FROM tableName.

In a similar manner, NOT EXISTS predicates are used to test if none of the records in a partition satisfy the specified condition. For instance, to find out which items would stay in high demand after the price increase, with probability > 50%, we could use the following query to compare the original ordered quantities with the hypothetical ones:

```sql
USING (SELECT *
FROM order_est AS oe, order_detail AS od
WHERE od.order_id = oe.order_id
AND od.item_id = oe.item_id)
PARTITION BY item_id
TEST WHETHER NOT EXISTS (od.qty > oe.hyp_qty)
WITH PROBABILITY > 0.5;
```

Note that, in both NOT EXISTS and EXISTS predicates, the test condition is optional; if unspecified, the test is applied on the contents of the isPresent attribute of the records in the output set of the USING query. Similarly, the PARTITION BY clause can be skipped; in that case, the entire output set of USING query is treated as a single partition.

The last category of hypothesis predicates, aggregation predicates, compute one or more SQL aggregate functions (such as SUM, AVG or COUNT) on all the records of each partition and test the results against a Boolean predicate. For instance, to determine if the total profits would increase after the hypothetical price change (with 90% probability), one could use the following query:
USING (SELECT *
FROM order_est AS oe, order_detail AS od
WHERE oe.order_id = od.order_id
AND oe.item_id = od.item_id)

TEST WHETHER
SUM(oe.price * oe.hyp_qty) > SUM(od.price * od.qty)
WITH PROBABILITY > 0.9;

3.1.1 Statistical Foundations

The problem of actually determining whether any of the partitions in a threshold query should be accepted has its roots in the testing of statistical hypotheses [99]. In a threshold query, each partition that is considered (such as the distinct item_id values in our first example) has an unknown probability \( \pi \) of satisfying the predicate under the TEST WHETHER clause. Given the clause WITH PROBABILITY > \( p \), the goal is to examine a set of \( N \) possible worlds or database instances to accurately choose between two hypotheses for each partition:

\[
H_0 : \pi = p - \epsilon \quad \text{or} \quad H_1 : \pi = p + \epsilon
\]

If \( H_0 \) is chosen, then the partition is labeled with \( res = \text{REJECT} \). In this case, the partition has been found to satisfy the predicate with a probability that is too small to meet the cutoff \( p \). If \( H_1 \) is chosen, then the partition is labeled with \( res = \text{ACCEPT} \).

In Section 2.2.1, it was stated that the kind of hypothesis test relevant to this work is a Binomial test wherein each observable data point is either 0 or 1. This is a consequence of the fact that the generative stochastic process for the data is embodied by a set of block-box VG functions that cannot be examined analytically; thus, for a given partition, the only available information is a pseudo-random bit string, where the \( i \)th bit in the string is a 1 if, in the \( i \)th Monte Carlo iteration or possible world,
the partition in question satisfied the predicate; the $i^{th}$ bit is a 0 if the partition did not. Since the hypothesis test can only look at the bit string and not inside of the VG functions, it is impossible to choose between $H_0$ and $H_1$ with total accuracy, and Type-I and Type-II errors are bound to occur with a certain frequency—that is, the label $res$ of a partition has probability $\alpha$ of being set to ACCEPT incorrectly (because $H_0$ is true), and a probability $\beta$ of being set to REJECT incorrectly (because $H_1$ is true).

In MCDB/SimSQL, $\alpha$ and $\beta$ are directly controlled by the user, via the following command-line entries:

\begin{verbatim}
   SET FALSE POSITIVE = 0.001;
   SET FALSE NEGATIVE = 0.001;
\end{verbatim}

Given a user-specified $(\alpha, \beta)$ pair, the number of iterations $N$ is controlled automatically by the system in order to guarantee that $\alpha$ and $\beta$ are respected. Since there is a direct relationship between $N$ and the running time of the query, almost all of the effort in the remainder of this chapter paper is targeted at determining whether $H_0$ or $H_1$ holds for a given partition using as few iterations as is possible.

### 3.2 Query Evaluation

As a useful point of reference, consider how “standard” MCDB/SimSQL—with the built-in ability to handle a threshold query directly—could be used along with an external hypothesis test to answer a threshold query.

Using the stochastic order_est table, imagine that a user wants to find those orders that had at least one item with a quantity of zero after the price increase. In standard MCDB/SimSQL, the user would first guess a value of $N$ that is large
enough to determine whether each order appears with high enough probability, and then issue the following SQL query:

```sql
SELECT DISTINCT order_id
FROM order_est
WHERE hyp_qty = 0;
```

Standard MCDB/SimSQL would process this query as depicted in Figure 3.1(a). The relations `params` and `order_detail` are passed to the `VGWrapper` operation, which uses them to realize the stochastic `order_est` relation by applying the `BayesDemand` VG function as specified in the stochastic `CREATE TABLE` statement (see Section 2.3.2). The resulting set of records/tuple bundles encodes the attribute `hyp_qty` in all $N$ Monte Carlo iterations. These bundles are then passed to relational selection and duplicate removal operations, and finally to the user.

To figure out if each `order_id` is present with high enough probability, the user would examine its corresponding `isPresent` attribute, count the number of $\top$ values as successful trials, and apply an appropriate statistical hypothesis test, which would either accept or reject each `order_id`—assuming that $N$ is large enough to guarantee the user-specified false positive and false negative rates.

While this approach could work, there are several problems with it. First, the user must choose and correctly apply an appropriate hypothesis test. Second, it requires a user to somehow know beforehand an appropriate value for $N$. Third, each and every partition uses the same $N$, even though it may be easy to accept (or reject) most partitions in only a few Monte Carlo iterations.

Now, imagine that this query was written directly as a threshold query:

```sql
USING order_est
PARTITION BY order_id
TEST WHETHER EXISTS (hyp_qty = 0)
WITH PROBABILITY > 0.95;
```
In this case, MCDB/SimSQL does not use a pre-defined $N$; instead, MCDB/SimSQL processes this query as a series of test blocks. In each test block, a relatively small, system-determined number of Monte Carlo iterations is run. The plan for the first test block of this query is illustrated in Figure 3.1(b). Just as in “standard” MCDB/SimSQL, $\text{params}$ and $\text{order_est}$ are passed to $\text{VGWrapper}$, which applies the $\text{BayesianDemand}$ function. However, the top of the query plan for a threshold query has two new operations: $\text{PartTest}$ and $\text{HypTest}$, which are described now.
3.2.1 The Partition-and-Test Operation

The input of the PartTest (short for “Partition and Test”) operation is the result set of the query under the USING clause. Given the clauses PARTITION BY A and TEST WHETHER φ, PartTest executes the following phases:

1. First, the input table T is divided into k partitions T₁, T₂, ..., Tₖ using the attributes in A, so that all the records in any given partition Tⱼ have the same attribute values for A. same attribute values for A belong to the same partition. If A = ∅, then all the records in T are placed in a single partition. If a stochastic attribute is in A, a Split operation is executed on T before partitioning.

2. Then, the partition test predicate φ is applied separately on each partition Tⱼ, returning an output record tⱼ that contains the attribute values for A that all the records in Tⱼ have in common, and tⱼ.isPresent with the result of φ.

The partition test predicate φ can take any of the following three forms: ANY, NONE or AGG.

ANY corresponds to the EXISTS φ category of predicates, applying the subordinate predicate φ (which, if unspecified, is always ⊤) to all the records in partition Tⱼ. The output is a single bit string b of length N, where

\[ b[i] = \bigvee_{t \in T_j} t[i].isPresent \land \varphi(t[i]) \]

Similarly, NONE corresponds to the the NOT EXISTS φ category. Here, φ is always ⊥ if unspecified, and

\[ b[i] = \bigwedge_{t \in T_j} t[i].isPresent \land \neg \varphi(t[i]) \]
As the name indicates, AGG corresponds to aggregate predicates, where the subordinate predicate $\varphi$ can be either

$$f(\cdot) \langle \text{op} \rangle \langle \text{const} \rangle \quad \text{or} \quad f(\cdot) \langle \text{op} \rangle g(\cdot)$$

where $f$ and $g$ are aggregate operations that evaluate arithmetic expressions over all the records in the partition $T_j$, each returning a single record $t_f$ (or $t_g$) with the result attribute $val$. For the sake of generality, let us assume that, in the case of an aggregate-to-constant comparison, $t_g.val \leftarrow \text{const}$ and $t_g.isPresent \leftarrow \top$. Thus, after the aggregate operation(s) have been executed, the output string $b$ is computed, where

$$b[i] = t_f[i].isPresent \land t_g[i].isPresent \land \varphi(t_f[i].val, t_g[i].val)$$

### 3.2.2 The Hypothesis Test operation

As the name indicates, the Hypothesis Test operation ($\text{HypTest}$ for short) encapsulates the application of a sequential hypothesis test as a relational algebra operator. $\text{HypTest}$ is parameterized with the probability threshold $p$ and takes as input the result set $T$ from the $\text{PartTest}$ operation, which contains the attributes $A$ identifying each partition and the results of the test predicate in the $isPresent$ attribute.

Moreover, $\text{HypTest}$ maintains state information in the temporary table $S$, which contains details about the partitions tested during the last test block (see Section 3.2.3). Each record contains the partitioning attributes $A$ and the attributes $numSucc$ and $numFail$, denoting the cumulative number of Monte Carlo iterations that passed and failed to pass the test predicate, respectively, throughout all the previous test blocks. Note that $S$ is empty during the first test block.
The actual hypothesis test is implemented as a library with the following publicly available methods:

- **Initialize**($\alpha, \beta, p, \epsilon, n_0, n_1$): sets up the basic parameters of the test. Here, $\alpha$ is the false positive rate, $\beta$ is the false negative rate, $p$ is the probability threshold, $\epsilon$ is a system constant (usually $10^{-6}$), and $n_0$ and $n_1$ represent the number of unsuccessful and successful trials from previous test blocks, respectively.

- **RunTest**($x$): runs the hypothesis test with the new Boolean observation $x$. Returns **ACCEPT** if the hypothesis has been accepted, **REJECT** if the hypothesis has been rejected, or **KEEP_GOING** if more observations are necessary to make a decision.

- **GetNum0**() and **GetNum1**(): return the total number of unsuccessful and successful trials so far, respectively.

- **GetEstN**: returns an estimate of the number of additional samples required to accept or reject the hypothesis.

**HypTest** operates on a single record a time. Algorithm 3.1 shows the sequence of steps that the operation undertakes for each record $t \in T$. If the hypothesis is accepted or rejected, then the partition corresponding to $t$ is sent to the user through the output set $T_{\text{user}}$. Otherwise, it is placed in the **KEEP_GOING** output set $T_{\text{KG}}$ and sent to the next test block so that more data can be generated.

### 3.2.3 Running Sequences of Test Blocks

Almost invariably, one or more partitions will fall into the **KEEP_GOING** category, and more Monte Carlo iterations will be required. When this happens, another test
Algorithm 3.1: HypTest(t)

Find $t' \in S$ such that $t.A = t'.A$

if $t'$ is found then
  Initialize($\alpha, \beta, p, \epsilon, t'.numFail, t'.numSucc$)
else
  Initialize($\alpha, \beta, p, \epsilon, 0, 0$)

$i = 1$

repeat
  $res = \text{RunTest}(t[i].isPresent)$
  $i = i + 1$
until $i = N$ or $res \neq \text{KEEP\_GOING}$

Create new $t^*$ with attributes $t.A$ and $res$

if $res = \text{KEEP\_GOING}$ then
  $T_{KG} = T_{KG} \cup \{t^*\}$
else
  $T_{user} = T_{user} \cup \{t^*\}$

Update $t'$ in $S$ with $t'.numSucc = \text{GetNum1}()$ and $t'.numFail = \text{GetNum0}()$

block is needed. The whole process of invoking the \texttt{VGWrapper} to create additional Monte Carlo trials, pushing the resulting records through the query plan, and then invoking \texttt{PartTest} and \texttt{HypTest} is performed again in the new test block.

Re-running the whole query plan from start to finish in the new test block is one possible solution. The problem with this approach is that the number of partitions in the \texttt{KEEP\_GOING} category will decrease every time that a new test block is run—thus, producing additional Monte Carlo iterations for every possible partition in every test block likely represents a waste of system resources. This is especially true given that one might expect the number of active partitions to decrease exponentially with respect to the number of test blocks that have been run, assuming that it is possible to obtain an \texttt{ACCEPT} or \texttt{REJECT} result for a relatively constant fraction of the active partitions in each test block.
Thus, MCDB/SimSQL uses a more targeted strategy, depicted in Figure 3.2. For each `KEEP_GOING` partition that is produced by the HypTest operator, HypTest attaches identifiers for all of the records in all of the base relations that could possibly contribute to the result for the `KEEP_GOING` partition. For example, if the HypTest operator does not have enough data to obtain an ACCEPT or REJECT result for the order with `order_id` 53 in our running example, it will send on the `KEEP_GOING`
output stream the record identifiers for all records in params and order_detail that could possibly influence the result for the order with order_id 53 (that is, it sends on the “lineage” of the KEEP_GOING partitions). A special relational operation called Route resides at the very bottom of the next test block and is used to split this lineage stream according to the base relation that each record identifier is associated with. The Route operation then routes the identifiers to a join with the appropriate base relation.

These joins effectively filter the base relations so that no base-relation-record that cannot contribute to some KEEP_GOING partition will enter the query plan. This means that, as the amount of partitions that are labeled as KEEP_GOING decreases, the amount of work associated with computing the test block decreases. This is a key innovation—one might expect that for most partitions, an accept or reject result is obvious and a few dozen Monte Carlo iterations will suffice. These partitions will all be removed from consideration after the first test block, resulting in significant savings.

3.2.4 Processing Records with Lineage

Every record that is processed by MCDB/SimSQL during evaluation of a threshold query includes an exhaustive list of all of the tables and records in those base relations that contributed to it. The purpose of this list is to guarantee that if a partition is labeled as KEEP_GOING, all of the records that could possibly have contributed to the partition will enter the query plan and be processed in subsequent test blocks. False positives are acceptable: processing extra records that do not contribute to a KEEP_GOING group will not cause problems, because the HypTest operation at the top of the next test block will ignore any partition that has not been labeled as
KEEP_GOING. On the other hands, false negatives are not acceptable, since failing to process a record that contributes in some way to some keep going partition can result in an incorrect result.

When processing threshold queries in MCDB/SimSQL, information about which base-relation records contributed to a record that is moving through the query plan is represented as a set of \((\text{tableID}, \text{recordID})\) pairs. In the database literature, such a list is usually referred to as the records lineage [54]. When a record is first scanned from disk, the only lineage information it contains is a single pair representing its original source relation and a unique record identifier for that relation. As query evaluation progresses, additional lineage information is accumulated. For example,

- A relational join operation \(T \leftarrow U \bowtie V\) which produces the record \(t \leftarrow (u \cdot v)\) will include in \(t\) all of the \((\text{tableID}, \text{recordID})\) pairs that are present in either \(u\) or \(v\) (removing any duplicate pairs).

- A \text{VGWrapper} operation that produces record \(t\) will append to \(t\) all of the pairs present in any record that contributed to \(t\). In \text{VGWrapper}, a record could contribute to \(t\) either by parameterizing the VG function that produced \(t\) via an inner query, or by contributing to \(t\) through the outer input relation.

- A grouping or aggregation operation that produces the record \(t\) appends to \(t\) all of the pairs present in any record that belong to its group or partition.

Since the lineage information can grow arbitrarily large (imagine, for example, that a billion-record table is aggregated via a \text{SUM} aggregate function down to a single number), MCDB/SimSQL imposes a maximum number of pairs from a single base relation that may be present in the lineage of a given record. If the number of pairs exceeds this threshold, all of the pairs are reduced down to a single pair.
with a special ALL value for recordID. Naturally, use of ALL may allow extra (unneeded) records to enter the query processing system during the next test block, but as discussed above, false positives are only problematic in the sense that they increase the number of records that will be processed by subsequent test blocks.

**Explicit Lineage vs. Inferred Lineage**

In the database literature, there are two methods that have been suggested for managing lineage: explicitly (as is done when processing threshold queries in MCDB/SimSQL), or by using algorithms that infer the lineage of a record, tracing it backward through a query plan [55]. The latter method is more pervasive in the literature, but there are two reasons that it is not used here. First, as discussed by Cui and Widom [55], explicit lineage is likely to be preferred when lineage tracing happens all of the time (as in MCDB/SimSQL threshold queries), rather than periodically. Second, the MCDB/SimSQL V GWrapper operation, which is a “black box” operation that encapsulates an arbitrary stochastic model through which lineage cannot be inferred.

**3.2.5 The Lineage Routing Operation**

When a test block completes, the records that have been placed in the KEEP_GOING set by the HypTest operation contain all the lineage information needed for correctly generating additional Monte Carlo iterations for any KEEP_GOING partition. The Route operation takes each of the records in the KEEP_GOING set and transforms their lineage information into a set of “lineage tables” that are joined with the base tables to filter out any records that do not contribute to a KEEP_GOING partition.

Given the set of base tables $B_1, B_2, \ldots, B_k$ that participate in the query, Route
Figure 3.3: Application of the lineage routing operation in a threshold query.

starts by creating the corresponding lineage tables $L_1, L_2, \ldots, L_k$, all with schema $(\text{recordID})$. Then, Route processes each record $t$ from the KEEP_GOING table. For each $(\text{tableID}, \text{recordID})$ pair in $t$’s lineage list, Route finds the lineage table $L_i$ corresponding to $\text{tableID}$ and inserts a single record with $\text{recordID}$ as the attribute value, ensuring that no other record in $L_i$ has the same value. If $\text{recordID} = \text{ALL}$ is seen, no additional records are recorded in $L_i$ and $B_i$ is marked as a no-filter base table.

After all the lineage tables have been generated, an equi-join operation between each base table $B_i$ that has not been marked as no-filter and the corresponding lineage table $L_i$ is executed to obtain a “new” base relation $B'_i$ for the current block. That is,

$$B'_i \leftarrow B_i \bowtie_{B_i.\text{recordID}=L_i.\text{recordID}} L_i$$

This results in preventing most of the records that do not contribute to some KEEP_GOING partition from being re-processed in subsequent test blocks. Figure 3.3 illustrates how Route is applied to the base tables $R$, $S$, and $T$; here, at least one of the partitions in
the \texttt{KEEP\_GOING} had the pair \((T, \text{ALL})\) in its lineage set, thus marking \(T\) as \textit{no-filter}.

### 3.2.6 Stochastic Considerations

At this point, the solution to the problem of avoiding computation associated with records that do not contribute to a \texttt{KEEP\_GOING} partition is almost complete: collect all of the lineage for every record that moves through the query plan, and then in subsequent test blocks ignore any tuples that were not present in the lineage of some \texttt{KEEP\_GOING} partition. However, this may produce incorrect results because, in some particular cases, it is possible that a tuple that did not contribute to some \texttt{KEEP\_GOING} partition during a given test block can contribute to a \texttt{KEEP\_GOING} going partition in any subsequent test block.

For example, consider the two tables \texttt{orders(order_id, customer_id)} and \texttt{customer(customer_id, city)}. Imagine that due to possible errors in the data loading process, \texttt{customer_id} has been modeled as a stochastic attribute, and for a given \texttt{orders} record, \texttt{order.customer_id} can actually take the value of one of several different customers whose \texttt{customer_id} is stored in the \texttt{customer} table. Now, imagine that the user asks threshold query that is concerned with the sales total of each city:

```sql
USING (SELECT *
FROM orders o, order_est oe, customer c
WHERE o.order_id = oe.order_id
AND o.customer_id = c.customer_id)
PARTITION BY c.city
TEST WHETHER SUM(oe.price * oe.hyp_qty) > 10000
WITH PROBABILITY 0.6;
```

To answer this query, it is necessary to compute the operation

\[
\text{orders} \bowtie_{o.customer_id - c.customer_id} \text{customer}
\]
which requires the application of a \texttt{Split} operation on \texttt{c.customer_id}. Suppose that \texttt{orders} contains the following records:

\begin{center}
\begin{tabular}{c|c}
1 & ACME \\
2 & ADVENT \\
3 & FOO \\
\end{tabular}
\end{center}

Also, assume $N = 2$ and that \texttt{customer} contains the following records during the first test block:

\begin{center}
\begin{tabular}{c|c|c}
  & AKME & AQME \\
ADVENT & ADUENT & HOUSTON \\
FEW & FOO & DALLAS \\
\end{tabular}
\end{center}

Since no record with a \texttt{customer_id} value of ACME is present in \texttt{customer}, the result of the join operation will not contain the first record from \texttt{orders}.

Suppose that the hypothesis test has labeled both cities as \texttt{KEEP\_GOING}. Then, in the next test block, the \texttt{customer} table is generated with the records:

\begin{center}
\begin{tabular}{c|c|c}
  & ACME & AGNE \\
ADVENT & ADVENT & HOUSTON \\
FOO & BOO & DALLAS \\
\end{tabular}
\end{center}

It is clear that the result set of the join operation between the above and \texttt{orders} should contain the first record from \texttt{orders}, which would have contributed to the \texttt{HOUSTON} partition. However, the lineage information for that record was lost during the previous test block, and the results of the threshold query would be incorrect.
The core problem here is the execution of an operation—a join in this example—that decided whether or not a record would be part of the output set by looking at the value of a stochastic attribute. In a particular Monte Carlo iteration, a record (and all of its lineage) can be filtered out by the join, but in the next iteration, the record may be accepted if the value of the stochastic attribute changes.

The solution to this problem is quite simple. When executing a threshold query, MCDB/SimSQL will extract all the lineage information from the input set of every Split operation and any selection or join that references a stochastic attribute. This lineage information is placed in a temporary freeze table \( F \). Then, in the next test block, both the contents of \( F \) and the \texttt{KEEP\_GOING} table are passed to the Route operation. This ensures, for example, that the lineage for the first record from \texttt{orders} will be placed in \( F \) before executing the join with \texttt{customer}, and that the record will be present in the next test block.

### 3.3 Experimental Evaluation

To test the validity of the techniques described in this chapter, five different queries from the TPC-H benchmark were tested using the original C/C++ implementation of the MCDB engine. All five queries were executed on a server machine with 8 CPUs and a scale-10 version of the TPC-H benchmark which amounted approximately 20GB of storage space for the base tables. Due to memory constraints, the number of Monte Carlo iterations per test block is limited to 60,000 for any given tuple bundle. Since, in some cases, the last remaining partitions from a query might require billions of Monte Carlo iterations to be classified as accepted or rejected, a maximum of 200,000 iterations is enforced and an arbitrary decision is taken if that number is exceeded by a partition.
3.3.1 Queries Tested

Query 3.1. This query uses a stochastic model from the original MCDB paper [10] to predict details about shipment durations, via the stochastic table `ship_duration`(sd_custkey, sd_when, sd_til_ship, sd_til_arr), where

- sd_custkey identifies the customer who initiated the shipment
- sd_when gives the date when the shipment was initiated
- sd_til_ship is stochastic and denotes the additional time from sd_when until the order is shipped
- sd_til_arr, also stochastic, denotes the time until the shipment arrives to the customer.

Given this stochastic table, we ask the question “Which customers who ordered today have at least a 20% chance of receiving part of their orders more than 20 weeks from now?” The SQL code for this query is:

```sql
USING (SELECT sd_custkey, sd_when + sd_till_arr AS len
FROM ship_duration
WHERE sd_when = today())
PARTITION BY sd_custkey
TEST WHETHER EXISTS (len > 140)
WITH PROBABILITY > 0.2;
```

Query 3.2. In this query, another model from the original MCDB paper is used to address a basic problem with the TPC-H data set: no price history for the various suppliers is recorded, and therefore it is impossible to know what the supplier cost of an item was in the past. Thus, a random walk backward from the current supplier price is performed to “guess” the supplier price at each month. This results in the stochastic table `price_hist`(ph_suppkey, ph_partkey, ph_month, ph_year,
ph_prc), where the stochastic attribute ph_prc denotes the price of the item identified by ph_partkey distributed by supplier ph_suppkey in the month ph_month of year ph_year.

Given this stochastic table, we ask the question “For which 1995 orders is there a 90% chance that we could have saved at least 20% if we had bought from the least expensive supplier?” The SQL code for this query is as follows:

```sql
USING (SELECT *
FROM price_hist, orders, lineitem,
(SELECT ph_partkey AS partkey,
   ph_month AS mo,
   ph_year AS yr,
   MIN(ph_prc) AS best_price
FROM price_hist
GROUP BY ph_partkey, ph_month, ph_year) AS best
WHERE o_orderkey = l_orderkey
AND l_suppkey = ph_suppkey
AND l_partkey = best.partkey
AND ph_partkey = best.partkey
AND best.mo = month(o_orderdate)
AND best.year = 1995
AND ph_year = 1995)
PARTITION BY o_orderkey
TEST WHETHER SUM(ph_prc-best_price) / SUM(ph_prc) > 0.2
WITH PROBABILITY > 0.9;
```

Query 3.3. The stochastic model for this query, estimates the number of items of a certain product and supplier that will be sold in the next thirty days.

To do this, each ordered line item from the previous year is re-created with a stochastic quantity.

Let $M$ be a random variable following a Poisson distribution with parameter $\lambda = \frac{30}{365}$. Given the quantity $q$ in a line item, the new stochastic quantity for that line item is estimated as $q' = qm$ where $m$ is an observation drawn from $M$. Then, the total amount of purchased products within the next month can be estimated by summing the value of $q'$ for all the line items of the same product and supplier.
The SQL code for creating this table would be as follows:

```sql
CREATE TABLE est_sells(partkey, suppkey, total) AS
  FOR EACH l IN (SELECT *
                 FROM lineitem
                 WHERE year(l_shipdate) = 1995)
  WITH mult AS Poisson(30/365.0)
  SELECT l_partkey, l_suppkey,
         SUM(l_quantity * mult.value)
  FROM mult
  GROUP BY l_partkey, l_suppkey;
```

Then, we ask the question “Which suppliers may run out of some product in the next 30 days, with 20% probability?” This is done by subtracting the available quantity of each supplier and item stored in the partsupp table and then executing the threshold queries:

```sql
USING (SELECT (ps_availqty - es.total) AS diff
         FROM partsupp, est_sells AS es
         WHERE es.partkey = ps_partkey
         AND es.suppkey = ps_suppkey)
PARTITION BY ps_partkey, ps_suppkey
TEST WHETHER EXISTS (diff < 0)
WITH PROBABILITY > 0.2;
```

Query 3.4. In order to test the performance of the freezing table mechanism, this query includes a join on a stochastic attribute. The model, obtained from [12], is designed to treat errors in the data set by defining the table likely_cust(orderkey, custkey) where the stochastic attribute custkey contains a guess of the “real” identifier of the customer in each order. We ask: “Which market segments of Japanese customers accounted for $1 billion in sales in a single month, grouped by month?”
Query 3.5. The stochastic model for this query, also from [12], assumes a correlation between shipping time and the probability that an item is returned by the customer, captured by a logistic model. The table proj_orders(orderkey,usual,half) contains the zero/one stochastic attributes usual and half, denoting whether an item is returned under two different conditions: “business as usual” and “shipment time cut by 50%.”

Given the above model, the following threshold query tests whether the expected change in revenue that will be observed by cutting the ship time in half will exceed 10%, with 30% probability:

```
USING (SELECT *
    FROM proj_orders AS po, lineitem
    WHERE po.orderkey = l_orderkey)
PARTITION BY o_custkey
TEST WHETHER (SUM(l_extendedprice*(1.0-l_discount)*half)
    - SUM(l_extendedprice*(1.0-l_discount)*usual))
/ SUM(l_extendedprice*(1.0-l_discount)*usual)
> 0.1
WITH PROBABILITY > 0.3
```

3.3.2 Results

Table 3.1 contains a summary of the results for each query, comparing the SPRT and End-Biased Test. The total number of query test blocks for each query and type of test is shown, alongside the total number of observations generated by all the test
Table 3.1: Summary of experimental results for threshold queries 3.1-3.5, with a comparison between the End-Biased Test and the SPRT.

<table>
<thead>
<tr>
<th>Query</th>
<th>Test blocks</th>
<th>Total observations</th>
<th>Time (HH:MM:SS)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SPRT</td>
<td>End-Biased</td>
</tr>
<tr>
<td>3.1</td>
<td>8</td>
<td>6</td>
<td>$3.6 \times 10^8$</td>
</tr>
<tr>
<td>3.2</td>
<td>22</td>
<td>15</td>
<td>$9.1 \times 10^9$</td>
</tr>
<tr>
<td>3.3</td>
<td>36</td>
<td>11</td>
<td>$6.0 \times 10^{10}$</td>
</tr>
<tr>
<td>3.4</td>
<td>6</td>
<td>4</td>
<td>$2.7 \times 10^8$</td>
</tr>
<tr>
<td>3.5</td>
<td>9</td>
<td>8</td>
<td>$3.3 \times 10^9$</td>
</tr>
</tbody>
</table>

Figure 3.4: Number of partitions (a) and wall-clock running time in seconds (b) per test block, for Query 3.1.

blocks, and the wall-clock running time from the beginning of the first test block until the last one finishes. The number of observations generated in a given test block is
calculated by multiplying the number of Monte Carlo iterations by the number of records produced by the \texttt{VGWrapper} operation.

Figures 3.4 to 3.8 show, for each query, the number of partitions at the beginning of each test block next to the wall-clock running time of the corresponding test block. Note that these figures only include the values corresponding to the End-Biased Test.

### 3.3.3 Discussion

The experiments seem to show that the methods described in this chapter are able to intelligently and dynamically adjust in order to accept or reject a large number of partitions early-on during the computation. Consider Query 3.3. The End-Biased Test only requests 54 Monte Carlo iterations in the first test block, and yet it is able to accept or reject 83,550 of the 100,000 partitions in the computation. It is true
that the number of Monte Carlo iterations per test block steadily increases, meaning that the amount of work per partition per block steadily increases, but there are few partitions that ever make it past the first block. For example, in Query 3.3, by the fifth test block only 761 partitions remain, which is less than 1% of the original total. As a result, the amount of time required per test block rapidly decreases—from 2,113 seconds in the first test block to 112 seconds in the second.

The one exception is Query 3.4, for one main reason. Neither the ETL-error model nor the TPC-H data generated differed across the five partitions, in the sense that any sort of skew or bias in parameter values was introduced. Since each partition has a large amount of data, due to the law of large numbers it is not surprising that the statistical properties of the five partitions are then very similar—hence all five partitions are eliminated as a group, after the same amount of work. This underscores
an unsurprising fact: the machinery described in this chapter is most useful when there are some partitions that can be rejected or accepted much earlier than others.

Let us consider a final point regarding this sort of “built-in” hypothesis test, and compare it to the obvious option of simply running “standard” MCDB/SimSQL and using a hypothesis test after the fact. A direct comparison of these two options is difficult, because running enough Monte Carlo iterations for the entire data set is very hard, but it is possible to make an educated guess as to how long it would take:

1. For each query, execute “standard” MCDB/SimSQL for one single Monte Carlo trial to obtain a time $t_1$,

2. Similarly, execute “standard” MCDB/SimSQL for 100 Monte Carlo trials to obtain a time $t_{100}$,
3. Make a rough guess as to the running time for $n$ trials using the following formula: $t_1 + \frac{n}{100}(t_{100} - t_1)$.

The reason for the above formula is that MCDB/SimSQL has certain high, fixed costs that are the same, no matter whether a single trial or one million are run—this includes computing and applying parameterizations, running relational operators, and so on. MCDB/SimSQL also has costs that scale roughly linearly with the number of Monte Carlo trials—this includes moving around tuple bundles, running the underlying random number generator, running selections on stochastic attributes, etc. Therefore, $t_1$ is a reasonable estimate for the former, and $\frac{1}{100}(t_{100} - t_1)$ a reasonable estimate for the latter.

Also, note that for 3.1 through 3.5, the SPRT required 21,403, 9,171, 16,102, 6,579 and 28,300 Monte Carlo iterations to finish all test blocks, respectively. Thus,
assuming the formula given above, it is possible to estimate the time in seconds required for “standard” MCDB/SimSQL to accept or reject each and every partition as 17,558, 95,046, 155,490, 10,095, and 51,074, respectively. These times are clearly greater than both the SPRT and End-Biased times given in Table 3.1, and underscore the utility of the methods presented in this work.
Chapter 4

History-aware Query Optimization

This chapter explores the idea of using the information present in a historical query workload within a cost-based query optimizer as a guide for generating alternative plans that have the side effect of materializing potentially useful result sets. For example, consider the following SQL query over the TPC-H schema.

```sql
SELECT AVG(ps_supplycost), p_name
FROM part, partsupp, supplier, nation
WHERE p_partkey = ps_partkey
AND ps_suppkey = s_suppkey
AND s_nationkey = n_nationkey
AND n_name = 'JAPAN'
GROUP BY p_name;
```

A conventional logical query optimizer is likely to favor a plan that filters the table nation first, joins it with supplier, and then joins the resulting table with partsupp before the final join with part and subsequent aggregation. However, in the presence of a history showing that variations of this query are executed frequently using different filtering predicates (such as another value for n_name), it would be preferable to use a plan that joins partsupp and supplier and part first, and then joins the resulting table (which is left as a materialized view after the query is finished) with the filtered nation. Even though this plan could be considered suboptimal in isolation, subsequent queries would benefit from the newly created materialized view.
This chapter describes a query optimizer called Hawc (History aware cost-based optimizer) that can take into account the value of intermediate results that it produces in this way. As new queries are issued, the optimizer tries to re-use previously-produced intermediate results, at the same time it takes into account the hypothetical benefits of materializing the intermediate results of a given plan. Individually suboptimal plans could be considered for execution if the history provides enough evidence that the benefits of generating certain intermediate results now and re-using them in the future outweigh the cost of running a more expensive plan now. These useful intermediate results are added to a space-restricted cache or “pool” that is optimized so that it contains a set of materialized views that maximizes the probability that future queries will match and re-use them.

Work on the Hawc optimizer is motivated by the design and development of the MCDB/SimSQL execution engine, which is distributed runtime running on top of Hadoop. During the design and development of MCDB/SimSQL, it became that if the system could automatically decide to archive useful intermediate results, it would represent a big win, especially given that MCDB/SimSQL is frequently tasked with executing various analytical queries, sequentially and one at a time, on a data set that tends to stay constant throughout the workload.

Further, MCDB/SimSQL's Hadoop-based execution environment is relevant. Hadoop clusters—especially those rented from cloud providers—tend to be constrained by RAM, CPU, and I/O bandwidth, but have excess, unused storage. For example, Amazon's “compute-optimized” c1.xlarge instance type has less compute power and RAM than a standard desktop machine, but has four disks and 1.6TB of storage. For most data-oriented computations, in order to obtain enough CPU and RAM, one finds it necessary to rent far more instances of this type than are strictly needed.
to store the data. Converting this storage to more computationally efficient query execution is desirable.

As the problem was considered, the following requirements for converting data storage to execution speed were developed:

1. Full integration with the optimizer, so that the optimizer could take into account the cost/benefit of using and of producing and archiving intermediate results.

2. Invisibility to the user, in that there are no outward signs of any recycling.

3. Incremental processing, so that all planning is done on a query-by-query (as opposed to batch) basis.

4. Opportunistic caching, in the sense that all materialized views that are created or used should be a by-product of running queries that the user wished to run anyway.

No existing solution meets these requirements. There is work on caching and re-using intermediate results [36–38, 40, 41, 48] but it does not meet requirement (1), in the sense that the optimizer is not aware of the potential benefits of intermediate results it produces, and cannot use information about positive side-effects when choosing a plan. There is the now-classical work on auto-tuning [43–46] which includes the idea of choosing materialized views to make processing a workload more efficient, but this work is complementary to the system described here, and does not meet requirements (1) to (4).

The remainder of this chapter describes a query optimization architecture that extends a cost-based logical query optimizer, integrating it with the analysis of query
history and creation and selection of materialized intermediate views. A cost computation model that employs history-based cost measures to account for the potential usefulness of the intermediate results produced by a query plan is also presented, together with an Integer Programming (IP) optimization strategy for selecting and pooling materialized views that have a high probability of being recycled by future queries, subject to disk space limits and other constraints.

### 4.1 Platform Overview

At the highest level, the Hawc optimizer attempts to make use of intermediate results from queries by maintaining two additional data structures, as illustrated in Figure 4.1: the history pool $\mathcal{H}$, and the view pool $\mathcal{V}$. These data structures are defined as follows:

- The history pool contains a set of recently-posed queries; for each query in the history pool, a set of alternative plans generated during optimization of the
query are maintained.

- The *view pool* contains a set of database tables that were created either as the final result or as the side effect of evaluating a previous query.

Given these data structures, when a new query \( Q \) is posed, the Hawc optimizer does the following:

1. First, \( Q \) is optimized. There are two key ways in which query optimization in Hawc differs from query optimization in a classical system. First, as in any database system that utilizes materialized views, the optimizer is able to use views from the view pool when choosing a plan for \( Q \). Second (and unique to Hawc) \( Q \) is optimized with respect to the *entire* history pool \( H \). That is, if a particular plan \( p \) for evaluating \( Q \) results in a new intermediate query result that could be used to evaluate another query in the history pool more efficiently than that query could be evaluated without that intermediate result, then the savings are credited to \( p \) when \( p \) is costed.

2. Once \( Q \) is optimized and executed, the view pool is updated. Given a particular plan \( p \), the optimizer may choose to save one or more intermediate results produced by \( p \). To choose how to update the view pool, Hawc produces an integer optimization program whose solution contains the set of views that will tend to result in the most inexpensive execution of the queries in the history pool, subject to the constraint that the total storage consumed by the view pool cannot exceed the maximum allocated by the system administrator.

3. Finally, the history pool is updated to reflect the newly-observed query \( Q \), so that intermediate results for subsequent queries that produce results that would help to execute \( Q \) more efficiently might be chosen.
In this way, the Hawc optimizer uses the set of queries that have previously been executed as indicative of the future workload, and the set of views in the view pool is constantly updated to reflect that workload.

The remainder of this section describes a few high-level details of the Hawc optimizer, starting with a discussion on how it calculates the cost of a given query plan, and then continue with a more detailed description of the history pool and the view pool.

4.1.1 Costing in the Optimizer

In the Hawc optimizer, there are two distinct phases of optimization: a logical phase (where a relational algebra expression for executing a query is constructed) and a physical phase where a physical implementation of that expression is constructed. The choice of whether and how to utilize and/or save intermediate results is made during the logical phase, and that is where this paper focuses. Thus, it does not consider implementation issues such as pipelining, sort orders, memory constraints, etc., in this paper.

That said, costing during logical optimization is quite flexible, in the sense that it is easy to apply a variety of different costing schemes within the framework, while allowing Hawc to potentially be used with a number of different execution backends (Hadoop, shared-memory multiprocessor, etc.).

This flexibility is maintained as follows. In the Hawc optimizer, each logical plan (that is, each relational algebra expression) has an associated cost vector \( c \). This is a list of statistics that describe various costs associated with the plan. The optimizer itself is agnostic to the meaning of each of the entries in the vector. In the simplest case, \( c \) might consist of a single number describing how many intermediate
tuples are materialized by the plan. \( c \) can, however, be arbitrarily complicated, listing the number of bytes read in at the leaves of the plan, the number of bytes (and tuples) processed by each of the different operation types, the number of UDF invocations, and so on. Then, the actual cost associated with that expression is \( r \cdot c \) for some constant, cost coefficient vector \( r \) (here, \( \cdot \) denotes the vector dot product). For the purposes of this discussion, it is not important to describe how \( r \) is obtained is tangential, only that it exists.*

The one fundamental assumption made by the optimizer is that plan costs are additive. More formally, it assumes that for a given plan \( p \), \( p.\Delta \) is a set of statistics describing the output of \( p \) (this might include, e.g., the number of tuples output from executing \( p \), the number of distinct attribute values for each output attribute, etc.). Then for a plan of the form

\[
p = \text{op}(p_1, p_2, \ldots, p_k)
\]

where each \( p_i \) is a sub-plan with cost vector \( p_i.\mathbf{c} \) and \( \text{op} \) is some relational operation, the cost vector for \( p \) is given by:

\[
p.\mathbf{c} = g_{\text{op}}(p_1.\Delta, p_2.\Delta, \ldots, p_k.\Delta) + \sum_{i=1}^{k} p_i.\mathbf{c}
\]

Here \( g_{\text{op}} \) is some cost function associated with the relational algebra operation \( \text{op} \). That is, the optimizer assumes that the cost vector describing \( p \) is itself a sum of the cost vectors of all of the sub-plans, plus the cost vector associated with executing the top-most operation \( \text{op} \). For example, consider the simplest case where \( p.\mathbf{c} \) contains

---

*In the actual implementation of the MCDB/SimSQL optimizer, \( r \) is computed statistically, using a linear regression model that learns \( r \) so as to accurately predict the cost of a training set of queries.
the number of tuples produced by the query plan. In the case where op is a join, \( \Delta \) would contain the number of tuples returned from the sub-operation, as well as the number of distinct values for the join attribute, and \( g_{op} \) would compute the standard formula for estimating the number of tuples resulting from a join [22].

The assumption that costs are additive is very useful, because it means that given that the cost of \( p \) is \( c = r \cdot p \cdot c \), one can quickly cost the alternative plan \( p' \) that results from replacing sub-plan \( p_i \) with \( p'_i \), using the formula:

\[
c' = c + r \cdot (p'_i \cdot c - p_i \cdot c)
\]

This simple formula is useful to consider the costs and benefits of replacing a plan for a particular subquery with a table scan over an intermediate result.

4.1.2 The History Pool

The history pool is simply a set of historical queries:

\[
\mathcal{H} = \{h_1, h_2, \ldots, h_n\}
\]

The history pool enables the optimizer to use recent queries to determine whether a plan for a new query might produce valuable intermediate results. As a new query is optimized, the intermediate results it produces are checked against the history pool to see how the queries in the history pool could utilize those results. Each historical query has two components:

- A weight \( w \) that indicates how important the query is to the history. During optimization of a new query, if a plan \( p \) produces an intermediate result that
could be used by a historical query $h$ to decrease its execution cost, that decrease is credited towards the cost of $p$. The weight $h.w$ is a multiplier that is used to discount (or increase) that credit. $h.w$ will typically decrease over time, as an older historical query is likely to be less representative of the expected workload than a newer one.

- A set of candidate plans $P$. This is used when a new query is optimized, to see how useful its intermediate results are when executing the historical query. One key aspect of the Hawc optimizer is that it does not store a single plan for a historical query in the history pool (the one that was actually executed, for example). It could do this; However, it will often be the case that the presence of a new intermediate result would render a plan other than $p$ as the optimal one for $h$. Thus, after a query is optimized, multiple, possible plans are recorded, and all of them are added to $h.P$. As a new query is optimized, its intermediate results are checked against all of the query plans in $h.P$ to see how helpful each candidate plan would be in executing $h$.

Note that $p.c$ and $p.\Delta$ are stored for each candidate plan in $p$, as well as for each sub-plan of each candidate plan in $p$, so that the optimizer can easily compute the utility of replacing any plan or any sub-plan with a scan of a candidate view that might be added to the view pool.

### 4.1.3 The View Pool

The view pool is a set of materialized views that were created by executing prior queries:

$$\mathcal{V} = \{v_1, v_2, \ldots, v_m\}$$
When a new query is executed, candidate plans are checked against the view pool to see if some view in the pool can be used to execute the plan more efficiently. One way in which the views stored in the pool differ from the sort of materialized views used by almost every modern database system is that they are not maintained in the presence of updates; when a view in the view pool becomes invalid, it is removed from \( \mathcal{V} \) and its physical realization is deleted from the system.

Each view in \( \mathcal{V} \) has the following components:

- A canonical plan \( p \) for the view. This is the plan (relational algebra expression) that produced the view, stored in a canonical form.
- The hash \( h \) of the canonical plan. This allows \( \mathcal{V} \) to be searched efficiently for useful materialized views during optimization.
- The set of statistics \( \Delta \) describing the view.
- The cost vector \( c \) giving the cost of performing a table scan over the view.

### 4.2 Example Query Workload

Before describing in detail how the history pool and the view pool are used during optimization, let us outline an example workload that contains several queries and views. This workload will be used to illustrate the optimization techniques that the Hawc optimizer uses.

Each query in the workload is an instance of one of the following two “template” queries, each using different selection predicates for filtering tuples based on the values of the attributes \( n\_name \) and \( s\_name \):
Query 4.1 ($Q_1$). Compute the average supply cost of all parts apportioned from suppliers from `nationName`.

```
SELECT AVG(ps_supplycost), p_name
FROM part, partsupp, supplier, nation
WHERE p_partkey = ps_partkey
AND ps_suppkey = s_suppkey
AND s_nationkey = n_nationkey
AND n_name = 'nationName'
GROUP BY p_name;
```

Query 4.2 ($Q_2$). Return the total number of returned parts of brand `brandName` for each supplier.

```
SELECT SUM(l_quantity), s_name
FROM lineitem, partsupp, part, supplier
WHERE l_suppkey = ps_suppkey
AND l_partkey = ps_partkey
AND ps_partkey = p_partkey
AND ps_suppkey = s_suppkey
AND l_returnflag = 'R'
AND p_brand = 'brandName'
GROUP BY s_name;
```

These queries are analyzed by the optimizer and used to enumerate definitions for materialized intermediate views. The following example views will be used:

View 4.1 ($v_1$).

```
SELECT *
FROM partsupp, part, supplier
WHERE ps_partkey = p_partkey
AND ps_suppkey = s_suppkey
```
Figure 4.2: Possible plans for Query 4.1 \((Q_1)\).

View 4.2 \((v_2)\).

| SELECT * 
| FROM partsupp, supplier 
| WHERE ps_suppkey = s_suppkey |

View 4.3 \((v_3)\).

| SELECT * 
| FROM lineitem, partsupp, supplier, part 
| WHERE l_partkey = ps_partkey 
| AND l_suppkey = ps_suppkey 
| AND ps_partkey = p_partkey 
| AND ps_suppkey = s_suppkey |

Figures 4.2 and 4.3 show four different plans corresponding to queries \(Q_1\) and \(Q_2\). The plans \(p_1\) and \(p_2\) are the optimal plans for \(Q_1\) and \(Q_2\), respectively. The alternative plan \(p'_1\) is a sub-optimal plan for \(Q_1\) that, if executed, would produce the
intermediate result sets for views $v_1$ and $v_2$. Similarly, the alternative plan $p'_2$ would produce the contents of view $v_3$ if executed.

The cost vector estimates for the example query plans are given in part (a) of Table 4.1. Plans with names such as $p_{ij}$ denote the sub-plan of $p_i$ that matches the definition of view $v_j$ and can be replaced with a table scan. Part (b) contains the table scan cost vector estimates and the disk space requirements for each view.

For simplicity, assume that the cost coefficient vector $\mathbf{r} = \mathbf{1}$ and that the history pool and view pool are initially empty. Nonetheless, the rest of this chapter will make use of the following example history (denoted as $\mathcal{H}_1$) to explain cost computation and view selection:
Table 4.1: Cost estimates for the example query plans and views.

<table>
<thead>
<tr>
<th>Plan</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>150</td>
</tr>
<tr>
<td>$p_2$</td>
<td>1,200</td>
</tr>
<tr>
<td>$p'_1$</td>
<td>450</td>
</tr>
<tr>
<td>$p'_2$</td>
<td>1,650</td>
</tr>
<tr>
<td>$p'_{11}$</td>
<td>400</td>
</tr>
<tr>
<td>$p'_{12}$</td>
<td>300</td>
</tr>
<tr>
<td>$p'_{21}$</td>
<td>400</td>
</tr>
<tr>
<td>$p'_{22}$</td>
<td>300</td>
</tr>
<tr>
<td>$p'_{23}$</td>
<td>900</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>View</th>
<th>c</th>
<th>$\Delta.s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>25</td>
<td>50</td>
</tr>
<tr>
<td>$v_2$</td>
<td>15</td>
<td>25</td>
</tr>
<tr>
<td>$v_3$</td>
<td>50</td>
<td>125</td>
</tr>
</tbody>
</table>

$h_1 = Q_1($JAPAN$)$,

$h_2 = Q_1($GERMANY$)$,

$h_3 = Q_1($UNITED STATES$)$,

$h_4 = Q_1($CANADA$)$,

$h_5 = Q_1($RUSSIA$)$,

$h_6 = Q_2($Brand#42$)$

Here, all queries have weight $w = 1$. The set of plans $P$ for queries $h_1$ through $h_5$ contains $p_1$ and $p'_1$ and their respective sub-plans, and the set of plans for $h_6$ contains $p_2$ and $p'_2$. 
4.3 Query Optimization in Detail

As a standard cost-based query optimizer, Hawc iteratively generates alternative query plans in order to obtain a plan of minimal cost. In addition to that, each alternative plan is inspected to determine which views could be created by executing it and materializing the intermediate results of some of its operations. The view descriptions gathered during this enumeration process allow the optimizer to search the view pool for equivalent materialized views that can be used to generate additional plans.

Moreover, computing the cost of each alternative query plan requires, besides calculating its cost vectors, matching its enumerated view descriptions with those of the queries in the history pool, so as to obtain the hypothetical cost of the history – that is, the cost of each historical query if the views enumerated for the plan had been materialized when the query was optimized.

4.3.1 View Enumeration and Matching

View enumeration is accomplished with a depth-first traversal of the plan that collects the elements that make up the canonical forms of some of its sub-plans†. Each canonical plan \( p \) has the following components:

- A set of attributes \( A \) present in each tuple of the output set of the root operation of \( p \).
- A set of base relations \( R \) used by the leaf operations of \( p \).

†To avoid enumerating large amounts of redundant and unpromising views, this process is restricted to non-Cartesian joins and aggregate operations with at least one grouping attribute.
• A set of predicates $\phi = \{\varphi_1, \varphi_2, \ldots, \varphi_k\}$ corresponding to the conjunctive expression $\varphi_1 \land \varphi_2 \land \ldots \land \varphi_k$ from the WHERE clause of the sub-query. This set can be partitioned as follows: $\phi_J$, which denotes all equi-join predicates; $\phi_E$, which denotes all semi-join and anti-join predicates; and $\phi_S$, which denotes the remaining selection and non-equi-join predicates. The predicates in $\phi_E$ are all of the form $(h_E, \varphi_E)$, where $h_E$ is the hash of the view that corresponds to the EXISTS sub-query and $\varphi_E$ is its join predicate.

• A set of grouping attributes $G$ used by aggregation or duplicate removal operations. Each attribute in $G$ must be present in the output set of the plan’s root operation.

Using the example from Section 4.2, the views $v_1$ and $v_2$ can be enumerated from plan $p_1'$. The canonical plan for $v_2$ would have the components:

$$A = \{\text{partsupp}.*, \text{supplier}.*\},$$

$$R = \{\text{partsupp}, \text{supplier}\},$$

$$\phi = \{\text{ps_suppkey} = \text{s_suppkey}\},$$

$$G = \emptyset.$$
has a fixed structure; for example, the operands of attribute comparison expressions are arranged in lexicographic order, so that $b > a$ becomes $a < b$.

Once the views for a given query plan have been enumerated, a transformation procedure in the optimizer will search the view pool for materialized views with matching canonical plans. If a match is found, the optimizer will generate an alternative query plan that replaces the corresponding operation and its sub-plan with a table scan on the matching view. Following the example from Section 4.2, if the view pool contains $v_1$ when $Q_1$ is optimized, a match with $p'_1$ would be found after the plan is generated, resulting in the alternative query plan

$$\sum(v_1 \bowtie_{\text{nation_id}} \sigma_{\text{n_name}}(\text{nation})).$$

A simple, straightforward matching procedure would take each enumerated view $v$ and search the view pool for a view $v'$ such that $v'.h = v.h$. In some cases, the optimizer also searches for other existing views that contain all the necessary columns and rows with the same duplication factor, such as views with additional cardinality preserving joins. To confirm that two views match, the optimizer checks for derivability by inspecting their canonical plans.

**View Derivability**

A view $v$ is said to be derivable\(^\dagger\) from another view $v'$ if its contents can be retrieved with a simple relational projection of the form $\pi_{p.A}(v')$ where $p = v.p$, $p' = v'.p$ and $p.A \subseteq p'.A$.

\(^\dagger\)Note that this notion of derivability is more restricted than those outlined in [36] and [31], which include relational selection—since the focus of Hawc is on cost computation, view matching is meant to be as unobtrusive and straightforward as possible.
To determine if derivability holds, it is necessary to inspect the canonical plans $p$ and $p'$ for the following conditions (Here, $\Leftrightarrow$ denotes logical equivalence).

1. All the attributes are projectable. That is, $p.A$ must be a subset of $p'.A$.

2. The selection predicates are equivalent. For each predicate $\varphi$ in $p.\phi_s$, there is one and only one $\varphi'$ in $p'.\phi_s$ such that $\varphi \Leftrightarrow \varphi'$.

3. The equi-join predicates are equivalent. For each predicate $\varphi$ in $p.\phi_j$, there exists a predicate $\varphi'$ in $p'.\phi_j$ such that $\varphi \Leftrightarrow \varphi'$.


5. The semi-join and anti-join predicates are equivalent. For each predicate $(h_E, \varphi_E)$ in $p.\phi_E$, there exists one and only one predicate $(h'_E, \varphi'_E)$ in $p'.\phi_E$ such that $\varphi_E \Leftrightarrow \varphi'_E$, and $h_E = h'_E$ or the corresponding views match.

6. The set of grouping attributes is the same. That is, $p.G = p'.G$.

The logical equivalence of the individual predicates involved in conditions (2) and (5) is determined by simple structural equivalence, directly comparing the operator and operands of any two predicates. A different procedure is applied to verify conditions (3) and (4), where transitivity relations are captured by using equivalence classes together with a primary key–foreign key graph algorithm, as described in [35].

**View Subsumption**

Given a query plan $p$ that has been successfully matched with views $v$ and $v'$, $v'$ is said to subsume $v$ with respect to $p$ if replacing one of the operations in $p$ with a table scan on $v'$ results in a new plan $p'$ that does not match with $v$. 
Note that, if a view $v'$ matches $p$, then every view $v$ that is derivable from $v'$ will also be subsumed with respect to $p$. Consider the example views $v_1$ and $v_2$ from Section 4.2: it can be shown that $v_2$ is derivable from $v_1$ according to the criteria described above (the additional relation supplier is joined with a primary key-foreign key predicate), and that $v_1$ subsumes $v_2$ with respect to $p'_1$.

As the rest of this chapter will show, knowledge of subsumption relations between views with respect to any given plan is necessary for calculating the cost of said plan and updating the history and view pools.

4.3.2 Costing with the History Pool

Each alternative query plan is evaluated for the potential that its enumerated views have for reducing the cost of the queries in the workload as described in the history pool. To this end, the basic cost model described in Section 4.1 is extended.

Recall that every plan $p$ is associated with a cost vector $c$ that results from adding the cost of the top-most operation in $p$ with the sum of the cost vectors of its sub-plans. The additivity of cost vectors allows the optimizer to efficiently calculate the cost of a plan after substituting one of its sub-plans with a table scan on a materialized view. Thus, the cost of the alternative plan $p'$ that results from replacing sub-plan $p_i$ with a table scan on the materialized view $v_i$ would be the following:

$$c' = c + r \cdot (v_i.c - p_i.c)$$

The above formula assumes that $v_i$ is in the view pool. As such, the cost vector $v_i.c$ was calculated using the statistics $v_i.\Delta$, which were obtained after the contents of the view were materialized. However, the goal is to compute the hypothetical cost
of the plans recorded for the queries in the history pool matched with views that may not have been materialized before, so $v_i, \Delta$ is probably unknown. In such cases, the optimizer assumes that the statistics for the sub-plan that is being replaced are accurate and can be used for computing the cost vector:

$$v_i, c = g_{ts}(p_i, \Delta)$$

where $g_{ts}$ is the cost function associated with the table scan operation.

**Hypothetical Query Costs**

Queries recorded in the history pool are always costed with respect to a given set of views. The value of this hypothetical cost is an approximation of the answer to the question “What would have been the cost of query $Q$ if the views in $V$ had been materialized when $Q$ was optimized?”

Since each query in the history pool has a variety of plans that are different in terms of the views that they are capable of matching, calculating the cost of a historical query amounts to selecting the plan that provides the minimum cost after matching and replacing a given set of candidate views. The cost of historical query $h$ with respect to the set of candidate views $V$ is defined as follows:

$$g_q(h, V) = \min_{p \in h.P} \{ g_p(V, p) \}$$

where

$$g_p(V, p) = r \cdot p'. c, \text{ given } p' = \text{CostReplace}(V, p)$$

Here, CostReplace (illustrated in Algorithm 4.1) returns the plan that results after
Algorithm 4.1: CostReplace$(V, p)$

\[ P = \{ p \} \]

\[ \triangleright \text{Match and replace views with } p \]

\textbf{foreach} \( v \in V \) where \( v.p \) matches \( p \) \textbf{do}

\[ p' = \text{tablescan}(v) \]
\[ p'.c = v.c \]
\[ P = P \cup \{ p' \} \]

\[ \triangleright \text{Match and replace views with } p \text{'s descendants} \]
\[ p' = p \]

\textbf{foreach} \( p_i \) \textbf{do}

\[ p'_i = \text{CostReplace}(V, p_i) \]
\[ p'_i.c = p'_i.c + (p'_i.c - p_i.c) \]
\[ P = P \cup \{ p'_i \} \]

\[ \triangleright \text{Return the best-cost replacement} \]
\[ p_{out} = \arg\min_{p^* \in P} (r \cdot p^*.c) \]

\text{return} \( p_{out} \)

matching and replacing an input plan \( p \) with a set of candidate views \( V \), starting at the root operation of \( p \) and greedily selecting the replacements that provide the lowest cost for each sub-plan \( p_i \). This heuristic also allows us to deal with subsumption between any two views matching with the plan by selecting only the replacement whose resulting sub-plan has minimum cost.

Let us return to the example laid out in section 4.2. Consider the hypothetical cost of query \( h_1 \) with the set of plans \( P = \{ p_1, p'_1 \} \) and the set of candidate views \( V = \{ v_1, v_2 \} \). First, CostReplace is applied on \( p_1 \), which returns \( p_1 \) unchanged since none of the views in \( V \) matches it; and then on \( p'_1 \), which decides for replacing \( p'_1; p_1 \).
over \( p'_{1}, p_{2} \) because of the smaller replacement cost:

\[
p'_{1}.c + (v_{1}.c - p'_{11}.c) = 75
\]
\[
p'_{1}.c + (v_{2}.c - p'_{12}.c) = 165
\]

With the cost vectors of all the plans for the query, compute \( g_{q}(h_{1}, \{v_{1}, v_{2}\}) = \min\{150, 75\} = 75 \).

**Final Plan Cost**

The hypothetical costs of all the queries in the history pool are factored into the final cost of each plan \( p \) that the optimizer generates, using the contents of the view pool and the views enumerated for \( p \) as parameters. Before it starts generating and costing plans, the optimizer processes each query \( h \) in the history pool \( \mathcal{H} \) by matching and replacing it with all the views in the view pool \( \mathcal{V} \), resulting in a new history \( \mathcal{H}^{*} \) where each query \( h^{*} \) contains the set of plans

\[
h^{*}.P = \{\text{CostReplace}(\mathcal{V}, p), \forall p \in h.P\}
\]

With this updated copy of the history pool, the optimizer proceeds to search for a plan of minimal cost. For each generated plan \( p \), Hawc enumerates the set of views \( V_{p} \) that would result from its execution as outlined in Section 4.3.1 and use them to compute the total hypothetical weighted cost for all the queries in the history and add it to the final plan cost:

\[
p.c' = r \cdot p.c + \sum_{h^{*} \in \mathcal{H}^{*}} h^{*}.w \times g_{q}(h^{*}, V_{p}).
\]
To put this in the context of the example workload from section Section 4.2, suppose Hawc is optimizing query $Q_1$ with the history described by $H_1$ and an empty view pool. Then, the optimizer generates plans $p_1$ and $p'_1$, enumerates their view sets $V_{p_1} = \emptyset$, $V_{p'_1} = \{v_1, v_2\}$ and calculates the final costs

$$p_1.c' = r \cdot p_1.c + g_q(h_1, V_{p_1}) + \ldots + g_q(h_6, V_{p_2})$$
$$= 150 + (150 + 150 + 150 + 150 + 150 + 1,200)$$
$$= 2,100$$

$$p'_1.c' = r \cdot p'_1.c + g_q(h_1, V_{p'_1}) + \ldots + g_q(h_6, V_{p'_2})$$
$$= 450 + (75 + 75 + 75 + 75 + 75 + 1,200)$$
$$= 2,025$$

Thus selecting $p'_1$ for execution.

**Cost Weight Policy**

Each query $h$ in the history pool is associated with a weight $h.w \in [0, 1]$ that defines how much its hypothetical cost contributes to the final plan cost.

The proof-of-concept implementation of Hawc described here uses a simple weighting policy where the most recent $k$ queries are given weight one, and the remaining queries have exponentially decreasing weight depending upon how long ago they were executed. That is, for a chronologically ordered sequence of queries $h_1, h_2, \ldots, h_n$ (where $h_1$ is the most recent query), $h_i.w = 1$ for $i \leq k$, and formula

$$h_i.w = 2^{-\frac{i}{k}}$$
for $i > k$.

4.4 Updating the Pools

After the runtime finishes executing a query, the history pool and view pool are updated with the details of that query. For the history pool, a new entry is appended with a collection of candidate plans and cost measures corresponding to the latest query, while the rest of the queries have their weights re-calculated according to the policy described in Section 4.3.2. Simultaneously, the intermediate result tables that were left materialized after executing the query are put together with their defining canonical plans and passed to the view pool.

4.4.1 History Pool: Selecting Plans

Each new entry $h$ appended to the history pool must contain a set of candidate plans $P$ that are generated by the query optimizer. The Hawc optimizer uses an A*-style search, where a set of the best plans encountered so far are maintained, and a set of rules are fired in sequence in an attempt to improve this set of plans. Search stops after either (a) a certain total number of plans have been considered, or (b) no active plan can be improved any more. Given that the number of candidate plans considered by the optimizer during this search can be huge (numbering in the tens of thousands) it is unfeasible to store the complete set of them as $P$.

Thus, the overall set of candidate plans must be cut down in order to form $P$. This plan selection consists of two stages. First, pre-selection is performed while the optimizer is searching for candidate plans in order to discard superfluous plans. Then, after the optimizer is finished, another algorithm reduces the number of candidate plans to a user-defined limit.
Pre-selection is a simple process, and relies on the fact that the optimizer always selects the candidate plan with the lowest cost after matching and replacing views. Given plans $p$ and $p'$ with costs $p.c < p'.c$, the costs of the plans resulting from applying CostReplace on $p$ and $p'$ will follow the same inequality if both plans match the exact same set of views with the same subsumption relations. Therefore, $p$ will always be preferred over $p'$ and keeping $p'$ in $h.P$ is unnecessary. Thus, as the optimizer is enumerating plans, it is possible to “forget” any plan $p'$ where there already exists such a plan $p$ in $h.P$.

Even using this pruning mechanism, once the optimizer has finished its search, there are still likely to be hundreds of plans that can be included in $h.P$; in practice, one wants to store only a dozen or so. Hence, Algorithm 4.2 is then executed in order to guarantee that the number of plans to be stored in the history pool never exceeds the user-defined maximum $k$.

Removing plans from $h.P$ affects workload cost computation and is likely to result in missed opportunities for view materialization, since some important view replacements could be lost. In order to minimize this effect, the greedy heuristic described by SelectPlans evaluates the diversity of view replacements in $h.P$ (defined as the total number of possible view replacements provided by all the plans) and how this diversity would be affected if a given plan were to be discarded, so as to avoid getting rid of plans that offer unique replacements not found in any other plan. Here, ViewReplace($p$) denotes the set of possible view replacement sets that can be achieved with plan $p$, where all the views belonging to the same replacement have no subsumption relations between them.

Consider executing the example query $Q_1$ described in Section 4.2. During pre-selection, no plans are discarded. Now, during SelectPlans we have ViewReplace($p_1$) =
Algorithm 4.2: SelectPlans($P, k$)

$P_{out} = P$

while $k < |P_{out}|$ do

$V = \bigcup_{p \in P_{out}} \text{ViewReplace}(p)$

$\Delta = \infty$

▷ Find plan $p^*$ with maximal diversity-$\Delta$

foreach $p \in P_{out}$ do

$V' = \bigcup_{p' \in P_{out} \setminus \{p\}} \text{ViewReplace}(p')$

▷ Calculate diversity-$\Delta$ for $p$ and compare

$\Delta_p = |V| - |V'|$

if $\Delta_p > \Delta$ then

$p^* = p$

$\Delta = \Delta_p$

$P_{out} = P_{out} \setminus \{p^*\}$

return $P_{out}$

$\emptyset$ and $\text{ViewReplace}\left(p'_1\right) = \left\{\{v_1\}, \{v_2\}\right\}$. Suppose $k = 1$, which means that one of the two plans must be discarded. Having $|V| = 2$, $\Delta_{p_1} = 2$ and $\Delta_{p'_1} = 0$, SelectPlans discards $p_1$ and keeps $p'_1$, since keeping $p_1$ would reduce the diversity of $h.P$ to zero, while keeping $p'_1$ leaves it unchanged.

4.4.2 View Pool: Selecting Views

Following the execution of each query, Hawc decides if the new materialized views created with the intermediate results of the query are to be discarded or kept for subsequent queries to use. The decision to accept a new view in the pool (or evict an existing one) is made by taking into account its disk space requirements and contribution to minimizing the cost of the workload that the history pool describes.

This procedure is modeled as an Integer Programming (IP) optimization problem that is generated after query execution. The objective of this program is to select the
plans and associated views that minimize the total cost of the workload as described in the history pool.

In order to curtail the size and complexity of the IP model, a preprocessing stage examines the view pool and evicts any views whose individual size exceeds the pool capacity, as they would be eliminated anyways. Subsequently, it generates a copy of the history pool \( \mathcal{H}^* \) that contains only the plans that make use of the views in \( \mathcal{V} \), also discarding any redundant plan that leads to a higher cost for its query while using the same view replacements as other alternatives.

**Objective Function**

Given the queries in the history pool \( \mathcal{H}^* \), the goal is to minimize the expression

\[
\sum_{h \in \mathcal{H}^*} h.w \times \hat{g}_q(h)
\]

Like the final cost function used in the query optimizer, the above expression denotes a total hypothetical cost for the history. There are, however, three key differences: the set of views being matched is an unknown, there is no current plan \( p \) being optimized, and a different function for the hypothetical query cost is used. This function \( \hat{g}_q \) is defined as follows

\[
\hat{g}_q(h) = \sum_{p \in h \cdot \mathcal{P}} \pi_p \times \hat{g}_p(p)
\]

where \( \pi_p \) is a binary variable used to determine if plan \( p \) is being used for computing the total cost. The function \( \hat{g}_p \) returns the cost of a given plan matched with all the possible materialized views from the view pool, including the ones that were created by the latest query. Let \( V_p \) denote the set of all possible pairs of the form \((v, p_i)\)
indicating that the view \( v \in \mathcal{V} \) matches sub-plan \( p_i \), thus

\[
\hat{g}_p(p) = r \cdot p.c + \sum_{(v, p_i) \in V_p} \mu_v \times (v.c - p_i.c)
\]

where \( \mu_v \) is a binary variable used to determine if view \( v \) is materialized. Accordingly, the purpose of the program is to find values for \( \pi \) and \( \mu \) that minimize the objective function.

**Problem Constraints**

Let \( S \) denote the maximum amount of disk space that the system administrator has allocated for the view pool. The following constraint is introduced to ensure that this limit is respected by the program:

\[
\sum_{v \in \mathcal{V}} \mu_v \times v.\Delta.s \leq S
\]

where \( v.\Delta.s \) is the amount of disk space required by view \( v \).

To ensure that the total cost is calculated correctly, each query \( h \) must select one and only one plan from \( h.P \). Therefore, it is necessary to introduce the constraint

\[
\sum_{p \in h.P} \pi_p = 1
\]

for each query \( h \in \mathcal{H}^* \).

Additionally, subsumption relations between the views that match a given plan must be taken into account, so that their corresponding replacement costs are not
added simultaneously to the same plan. The constraint

\[ \pi_p \times (\mu_v + \mu_{v'}) \leq 1 \]

is introduced for every pair of views \( v, v' \) that have a subsumption relation with respect to plan \( p \), for all \( p \in h.P, h \in \mathcal{H}^* \). The constraint guarantees that, if \( p \) is not the selected plan for the query, it is possible for both views to be materialized simultaneously, as long as they do not have a subsumption constraint with respect to another plan\(^6\).

After the IP program has finished, the optimizer examines the resulting variable \( \mu \) so that only the views for which \( \mu_v = 1 \) are kept in the view pool.

To illustrate this in the context of the example workload from Section 4.2, consider running this optimization program using the history \( \mathcal{H}_1 \) with an additional query \( h_7 \) corresponding to the recently optimized \( Q_1 \), which has left the contents of \( v_1 \) and \( v_2 \) materialized after execution. Suppose that the maximum amount of disk space for the view pool is \( S = 125 \) and that it contains view \( v_3 \) from the previous run. This setup would correspond to the objective function

\[ \min. 1 \times \hat{g}_q(h_1) + 1 \times \hat{g}_q(h_2) + \ldots + 1 \times \hat{g}_q(h_7) \]

\(^6\)Although multiplying two variables makes the problem nonlinear in general, this can be avoided when both variables are binary using surrogate variables and constraints. For the sake of clarity, those changes are not included in this formulation.
where $\hat{g}_q$ is as follows (Note that the equations for $h_1$ through $h_5$ and $h_7$ are the same):

$$\hat{g}_q(h_1) = \pi_{p_1} \times \hat{g}_p(p_1) + \pi_{p'_1} \times \hat{g}_p(p'_1)$$

$$\hat{g}_q(h_6) = \pi_{p_2} \times \hat{g}_p(p_2) + \pi_{p'_2} \times \hat{g}_p(p'_2)$$

and $\hat{g}_p$ is as follows:

$$\hat{g}_p(p_1) = 150$$

$$\hat{g}_p(p_2) = 1,200$$

$$\hat{g}_p(p'_1) = 450 + \mu_{v_1} \times (v_1 \cdot c - p'_{11} \cdot c) + \mu_{v_2} \times \ldots$$

$$= 450 + \mu_{v_1} \times (25 - 400) + \mu_{v_2} \times (15 - 300)$$

$$\hat{g}_p(p'_2) = 1,650 + \mu_{v_1} \times (v_1 \cdot c - p'_{21} \cdot c) + \mu_{v_2} \times \ldots$$

$$= 1,650 + \mu_{v_1} \times (25 - 400) + \mu_{v_2} \times (15 - 300)$$

$$+ \mu_{v_3} \times (50 - 900).$$

Subject to the space constraint

$$\mu_{v_1} \times 50 + \mu_{v_2} \times 25 + \mu_{v_3} \times 125 \leq 125$$

the query plan constraints

$$\pi_{p_1} + \pi_{p'_1} = 1$$

$$\pi_{p_2} + \pi_{p'_2} = 1$$
and the subsumption constraints

\[
\begin{align*}
\pi_{p_1'} \times (\mu_{v_1} + \mu_{v_2}) & \leq 1 \\
\pi_{p_2'} \times (\mu_{v_1} + \mu_{v_2}) & \leq 1 \\
\pi_{p_2'} \times (\mu_{v_1} + \mu_{v_3}) & \leq 1 \\
\pi_{p_2'} \times (\mu_{v_2} + \mu_{v_3}) & \leq 1.
\end{align*}
\]

Therefore, there are only three valid solutions to this program:

1. Keep \(v_3\), evict \(v_1\) and \(v_2\) (cost = 1,700)
2. Keep \(v_1\), evict \(v_2\) and \(v_3\) (cost = 1,650)
3. Keep \(v_2\), evict \(v_1\) and \(v_3\) (cost = 2,190)

and solution (2) is the preferred one.

4.5 Experimental Evaluation

In this section, an empirical evaluation of the Hawc optimizer is presented. The goal is to answer the following two questions:

1. Does Hawc’s history-driven approach to query optimization result in a significant improvement in workload execution time, compared to the more conventional strategy of caching and re-using the intermediate results of the best possible plan for each individual query?
2. Does Hawc’s view selection approach makes good use of restricted disk space, delivering performance comparable to (or better than) that of a LRU or FIFO cache replacement policy?
4.5.1 Setup

The high-level approach was to generate various workloads using the TPC-H benchmark database. Each workload contains 100 queries, where each query in the workload is a variant of one of the original 22 TPC-H benchmark queries. To make exact repetitions of the same TPC-H query unlikely, the selection predicates of each query in a workload were generated at random, within the appropriate ranges of possible values described for the benchmark.

To make the experiments realistic, the distribution of queries that are issued at the beginning of a 100-query workload are different than the set of queries issued at the end of a workload. The workload generation process is as follows:

1. Randomly generate two, 22-dimensional vectors $\mathbf{x}$ and $\mathbf{x}'$ by sampling from a Dirichlet distribution with $K = 22$ categories and concentration parameters $\alpha_1, \alpha_2, \ldots, \alpha_K = 0.1$. Both $\mathbf{x}$ and $\mathbf{x}'$ are then vectors of probabilities, where $\mathbf{x}$ lists the probability of each of the 22 queries being encountered at the very beginning of the workload ($x_i$ is the probability of seeing query $i$ at the start of the workload) and $\mathbf{x}'$ lists the probability of each of the 22 queries being encountered at the end of the workload.

2. Generate each query $Q_i, i \in \{0, 1, \ldots, 99\}$ by sampling from a Categorical distribution with event probabilities $\mathbf{p}_i = p_1^i, p_2^i, \ldots, p_K^i$ where

$$p_i^j = \frac{x_j \times (99 - i) + x'_j \times i}{99}.$$

In this way, the likelihood of encountering each query drifts from $\mathbf{x}$ to $\mathbf{x}'$ over time.
All runs were executed on an Amazon EC2 cluster of 5 m2.4xlarge machines, each with 8 relatively low-end virtual cores and 64GB of RAM. The TPC-H benchmark tables were generated with a scale of 250, which amounts to 500GB of base data distributed over HDFS so that the share of data for each machine approximates a scale 50 dataset.

4.5.2 Benchmarks Run

History-based optimization

To test the effectiveness of Hawc’s methods, five different workloads were generated as described above and executed separately. To run a workload, each one of its comprising queries were executed sequentially and one at a time, collecting their individual wall-clock running times. Each of the five workloads is run multiple times, using a different execution strategy at every repetition.

The three workload execution strategies are:

1. A conventional cost model that chooses the best plan for each query and runs each query independently without re-using any materialized intermediate views. At first glance, this approach may have little chance of outperforming the methods proposed in the paper. However, there is a hidden (but important) cost associated with these methods: like all modern database systems, MCDB/SimSQL makes extensive use of pipelining. If the optimizer forces the runtime to materialize an intermediate result for later use, it can have a significant, negative effect on the execution time. This option is termed “no materialization.”

2. Allow the optimizer to make use of cached intermediate results (in the way described in this paper), but do not allow the optimizer to alter the plans it
chooses in order to purposely create materialized views that might be useful for subsequent queries. Instead, any intermediate results that happen to be materialized to disk are automatically saved, and only those views can be used by the optimizer for executing subsequent queries. The view pool starts out empty, and it is allowed to grow to an unconstrained size. This method has the benefit of being able to use cached results, but it has advantage that it does not disrupt pipelining by forcing the execution engine to materialize intermediate results. This option is termed “query-based.”

3. Hawc’s full, history-based model with the possibility of running sub-optimal plans to materialized useful views, starting with an empty history and empty view pool. Again, the view pool is allowed to grow to an unconstrained size. This option is termed “history-based.”

Tables 4.2 and 4.3 summarize the results in terms of total workload execution time, amount of disk space and number of views materialized by each method.

Table 4.4 illustrates a cost/benefit comparison of the query-based and history-based methods.
<table>
<thead>
<tr>
<th>Workload</th>
<th>Query-based</th>
<th>History-based</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td># Views</td>
</tr>
<tr>
<td>W1</td>
<td>290GB</td>
<td>127</td>
</tr>
<tr>
<td>W2</td>
<td>221GB</td>
<td>124</td>
</tr>
<tr>
<td>W3</td>
<td>194GB</td>
<td>49</td>
</tr>
<tr>
<td>W4</td>
<td>216GB</td>
<td>81</td>
</tr>
<tr>
<td>W5</td>
<td>433GB</td>
<td>128</td>
</tr>
</tbody>
</table>

Table 4.3: Maximum total disk space occupied by the pool and total number of views created during the execution of each workload.

based methods with respect to the no materialization results, for all queries in the five workloads. This table shows the decrease in running time (HH:MM) for those queries that matched and recycled (“hit”) at least one intermediate result set from the view pool. Similarly, the table shows the increase in running time (HH:MM) for those queries that produced (“create”) at least one new intermediate result set and placed it in the pool. Also included are the maximum and average time increase/decrease, as well as the geometric mean of the corresponding speedup/slowdown of each query.

The plots from Figures 4.4 to 4.8 show the cumulative running time as a function of the fraction of the queries in the workload that have been executed so far, for each of the five workloads tested, and for each of the three optimization and execution strategies. As mentioned above, there are no view pool space constraints.

**View Pool Selection**

The second set of experiments tests the performance of the Integer Programming method described in Section 4.4.2, particularly in maintaining the functions of the
Table 4.4: Query-wide comparison of benefits and costs for the query-based and history-based methods, with respect to the “no materialization” benchmark.

<table>
<thead>
<tr>
<th></th>
<th>Query-based</th>
<th>History-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. View Pool Hits</td>
<td>253</td>
<td>331</td>
</tr>
<tr>
<td>Max. Time Decrease</td>
<td>00:47</td>
<td>01:28</td>
</tr>
<tr>
<td>Mean Time Decrease</td>
<td>00:16</td>
<td>00:22</td>
</tr>
<tr>
<td>Mean Hit Speedup</td>
<td>2.98</td>
<td>3.71</td>
</tr>
<tr>
<td>No. View Pool Creates</td>
<td>297</td>
<td>62</td>
</tr>
<tr>
<td>Max. Time Increase</td>
<td>00:04</td>
<td>00:12</td>
</tr>
<tr>
<td>Mean Time Increase</td>
<td>00:01</td>
<td>00:03</td>
</tr>
<tr>
<td>Mean Create Slowdown</td>
<td>1.02</td>
<td>1.13</td>
</tr>
</tbody>
</table>

Figure 4.4: Running time (HH) as a function of the fraction of executed queries in workload W1.

To do this, W2 is executed using the full, history-based model multiple times. Over the various runs, the view pool size constraint is changed, as well as the way...
in which the system chooses which intermediate results to include within the view pool. Let $S = 360$GB be the maximum size reached by the unconstrained view pool during workload execution using the history-based cost model. The benchmark has three sets of runs, where the maximum size of the view pool is allowed to be $\frac{2S}{3}$, $\frac{S}{2}$, $\frac{S}{4}$, respectively.
For each set of runs, three different replacement strategies are tested:

1. The IP-based formulation proposed here, with the cost weight policy described in Section 4.3.2, using $k = 20$,

2. A least-recently-used (LRU) caching policy, and
Table 4.5: Total workload running time (HH:MM) with different pool space constraints and caching methods.

<table>
<thead>
<tr>
<th>Max. view pool size</th>
<th>LRU</th>
<th>FIFO</th>
<th>Hawc IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>2/3S</td>
<td>17:33</td>
<td>25:22</td>
<td>18:53</td>
</tr>
<tr>
<td>1/2S</td>
<td>30:52</td>
<td>36:06</td>
<td>17:54</td>
</tr>
<tr>
<td>1/4S</td>
<td>39:58</td>
<td>34:40</td>
<td>19:35</td>
</tr>
</tbody>
</table>

3. A first-in/first-out (FIFO) caching policy.

For the latter two policies, views are removed from the pool, in sequence, until the total size of the view pool is under the maximum size allowed.

A summary of the total running time for all the different replacement strategies and view pool size constraints is shown in Table 4.5.

The plots shown Figures 4.9 to 4.11 illustrate the cumulative running time as a function of the fraction of the queries in workload W2 that have been executed so far, for each of the three replacement strategies tested, and for each of the three different constraints on maximum view pool size.

4.5.3 Discussion

The experiments seem to show that Hawc’s methods are capable of achieving significant workload speedups—ranging from to 1.4 to 3.5—given unconstrained disk space for storing materialized intermediate views. Even then, the maximum size of this unconstrained pool of views is only, in average, about two thirds of the size of the base dataset.

One interesting outlier from the first set of experiments is workload W3, for which the speedup was only marginal with respect to the query-based approach. The rea-
Figure 4.9 : Running time as a function of the fraction of executed queries with maximum view pool size $\frac{2S}{3}$.

Figure 4.10 : Running time as a function of the fraction of executed queries with maximum view pool size $\frac{S}{2}$.

son is simple: the majority of this workload happened to be composed of queries from the TPC-H benchmark that can extract very little benefit from having some of their sub-expressions materialized without all the selection predicates particular to an individual realization of the query, such as Q16 and Q12.

As shown in Table 4.4, the history-based costing methods lead to the creation
of relatively few views that are relevant to the needs of the workload, providing substantial improvements in query execution time, with acceptable overhead on the queries that execute suboptimal plans to create new views.

The results from the second benchmark show that there is a large amount of value to the IP formulation. As the space constraints become more stringent, there is not much degradation in workload execution time when the IP formulation is used to choose which view to evict from the pool. However, both the LRU and FIFO replacement strategies take about twice as much time to run when the size of the view pool is reduced by a factor of four. It is interesting that there is some variation in the results; for example, LRU is actually the best policy at a size of $\frac{S}{3}$. This is due to the fact that in the end, anticipating the future query workload is an imprecise task. In the case where LRU performed better, after the 33rd query, the IP decided to drop a large view that happened to be usable by the 67th query, while the LRU methodology decided to keep it. However, while LRU and FIFO can certainly happen to get lucky, in the more restricted cases, far more choices must be made, and the
more careful strategy employed by the IP formulation seems to do far better.
Chapter 5

Natural Specification of Stochastic Models

Multivariate probability distributions are prevalent in contemporary stochastic models, particularly in the machine learning literature. These stochastic models are often described using matrix and vector operations, and their implementations are often carried out in languages that provide native support for structured, array-based types, such as R, MATLAB, or BUGS [16].

Representing vectors and matrices as tables in a relational database is a fairly straightforward task. However, implementing linear algebra operations such as matrix and vector multiplication using SQL and relational algebra is cumbersome and error-prone. Consider, for example, the Gibbs sampler for the Bayesian Lasso [18], described in Appendix A.1. As shown, implementing this model requires approximately 100 lines of code using the MCDB/SimSQL extensions to the SQL language. Most of this code is comprised of relational descriptions of matrix and vector operations, which require the use of nested sub-queries, joins and group-by aggregation operators.

Consider the sum of squares

$$(\tilde{y} - X\beta)\top (\tilde{y} - X\beta)$$

Assuming a schema with the base data tables response(respID,respValue) and regressor(respID,regID,regValue) the above expression would be im-

...
SELECT SUM((cenY.val-XB.regValue)*(cenY.val-XB.regValue))
FROM
(SELECT x1.i AS i, SUM(x1.val * b.val) AS sumVal
FROM regressor AS x1, beta[i] AS b
WHERE x1.j = b.j
GROUP BY x1.i)
AS XB,

(SELECT y1.i AS i, (y1.respValue - my.meanVal) AS val
FROM response AS y1,
(SELECT AVG(y2.respValue) AS meanVal
FROM response AS y2
) AS my
) AS cenY

WHERE XB.i = cenY.i;

Here, the nested sub-query cenY is used to compute the centered response vector \( \hat{y} \), and the nested sub-query XB is used to compute the centered response vector \( X\beta \). To write the above code, a user must decide the order of matrix and vector operations and devise a way to decompose the matrix and vector operations so that each may be translated into relational sub-queries, and join them together to produce the final result. This procedure is cumbersome, error-prone, and results in code that is hard to understand and maintain.

Furthermore, making use of atomic vector and matrix types, although helpful (particularly when it comes to improving the performance of linear algebra operations [107]), does not result in an entirely natural and straightforward syntax, and the user must still decide the best representation for each structure and the order in which linear algebra operations are to be applied.

The lack of direct support for ordered structures and their difficult manipulation has been identified as a key reason behind the limited acceptance of relational databases in scientific applications [19]. Still an ongoing research topic, array DBMSs...
are concerned with supporting array structures in a database environment, either as first-class objects such as tables [68, 70, 71], or as second-class objects such as attributes [69], and the implications of adding support for such types concerning query languages and algebra [72–74].

The objective of this chapter is to provide a description for a high-level notation and language to represent stochastic models in a natural, succinct way. With this language, an end-user may specify how the random variables are sampled, in a similar fashion to that of R or BUGS*. The level of abstraction of this language must be higher than that of SQL, making implicit all operations between structured types such as vectors, matrices, multi-dimensional arrays and sparse structures. Ultimately, the model must be translated into relational algebra or SQL, so that it may be executed by the MCDB/SimSQL runtime.

This notation and language for describing stochastic models must meet the following requirements:

1. The level of abstraction must be high enough for random variables to become first-class objects that are assigned to a distribution function for sampling.

2. The expressive power must not exceed that of SQL and relational algebra as supported by SimSQL/MCDB. Models that cannot be translated into SQL should not be allowed in the syntax and semantics.

3. Data independence between structured types used by the model and their relational representations must be guaranteed.

*Note that, in BUGS, users specify the generative model, and the BUGS runtime derives the MCMC sampler for inferring the parameters of the model. The notation I propose is for describing the MCMC sampler once it has been derived by a third party.
4. Ordered structures, such as arrays of arbitrary dimensionality, must be supported as first-class objects— that is, as variables. Matrix and vector algebra operations must be implicit too.

5. Set-based structures, such as tables and sparse arrays, must also be supported as variables and integrated seamlessly with the rest of the notation.

The rationale behind requirement (5) is that dense structures such as matrices, vectors and other multi-dimensional arrays are not enough for accurately describing data objects in several classes of machine learning models. Take, for example, Latent Dirichlet Allocation [108] (LDA), a standard text mining model (the Gibbs sampler is described in Appendix A.2). In the LDA model, the data set contains a large multi-dimensional structure \( W \) where \( W_{i,j} \) denotes the times word \( w_j, j \in \{1, 2, \ldots, m\} \) appears in document \( d_i, i \in \{1, 2, \ldots, n\} \). Although it is possible to represent \( W \) as a matrix with \( n \) rows and \( m \) columns, doing so results in a highly sparse matrix, because the number of distinct words that make up any given document is usually in the dozens, representing a small fraction of the size of the dictionary. Therefore, a different kind of abstraction for representing sparse structures in the language is necessary.

Previous work focused on the integration of ordered structures into query processing began with the development of the Nested Relational Calculus for Arrays [72] which allowed for a high-level query language based on the syntax of comprehensions [76]. Some of the advantages of the syntax of comprehensions are that it allows for elegantly describing collections such as arrays and sets as queries, its expressive power does not exceed that of relational algebra or SQL [77], and that the translation process is relatively straightforward, as shown by approaches like the RAM
algebra [74, 75].

For the reasons specified above, comprehension syntax expressions are an important component of the language described in this chapter. However, the purpose of this language is not the description and evaluation of single queries over arrays, but the description of entire models containing multiple array-like structures whose contents are determined by the evaluation of queries that are expressed in comprehension syntax. Furthermore, those queries are described in mutually-recursive fashion and require extensions on the target language for their correct translation—that is, the MCDB/SimSQL flavor of SQL with table indices for describing Gibbs sampler.

Another important feature of the language is that its compilation and translation algorithms must consider the multiple possible representations of an array or ordered structure in a relational context, which may be challenging in a system that permits the usage of attributes with vector or matrix types between such attributes †, alongside multiple possible implementations for functions and operators on those types. Thus, the translator must execute additional optimization steps to determine which representation and relational implementation is the most efficient.

The remainder of the chapter is organized as follows. Section 5.1 describes the syntax and semantics of the language. Section 5.2 summarizes the compilation and translation process. Section 5.3 details how models are checked and normalized after being parsed. Section 5.4 details the methods I employ for generating the schemas and queries for executing the model in MCDB/SimSQL. Section 5.5 describes the implementation of the language and the models and tests that were applied to verify the correctness of the implementation.

†Recently, MCDB/SimSQL has been extended with this capability, including user-defined and VG functions that take vectors and matrices as input parameters and produce them as output values.
5.1 Syntax and Semantics

The basic programming unit of the language and notation is the model. The general syntax for a model is:

```plaintext
model {modelName} {
  data {
    {elementDefinition1}
    {elementDefinition2}
    ...
  }
  var {
    {elementDefinition1}
    {elementDefinition2}
    ...
  }
  init {
    {modelStatement1}
    {modelStatement2}
    ...
  }
  {modelStatement1}
  {modelStatement2}
  ...
}
```

A model consists of one or more data elements, and each element is associated with a structural definition—that is, a description of the element as a data type. Data elements can be classified as constant data elements and variable data elements.

Constant data elements, located in the data section, describe the base data set, together with ancillary constant values that the model requires, such as hyper-parameters. For example, in the Bayesian Lasso, the regressor matrix $X$, the response vector $y$ and the Lasso parameter $\lambda$ would be represented as constant data elements.

On the other hand, each variable data element placed in the var section is also associated with a definition that describes the set of mathematical operations that determine the value of the element. These definitions are known as model statements.
In the language.

In the Bayesian Lasso, the vectors of regression coefficients $\boldsymbol{\beta}$ and features $\boldsymbol{\tau}$ and the variance $\sigma^2$ would be represented as variable data elements, defined as data types in the `var` section and with the correspondent model statements that determine how their values are computed.

The `init` section is required for stochastic models that generate samples cyclically and whose variables require an initialization value or statement for the “zeroth” generation, such as Gibbs sampling models.

### 5.1.1 Data and Variable Element Definitions

A data or variable element declaration binds a symbol name to a data type that determines the structure of that element. The general syntax for a definition is

$$
\langle \text{elementName} \rangle : \langle \text{elementType} \rangle ;
$$

where `elementName` is the symbol name and `elementType` denotes the symbol’s corresponding data type. Data types can be categorized as *atomic* types, *domain* types and *compound* types.

Atomic types define elements containing a single value and can be either are `integer`, `real`, or `string`. For example, in the Bayesian Lasso, the constant $\lambda$ is represented as a `real` number:

```
data {
    lambda: real;
    ...
}
```

Domain types can only be used when declaring constant data elements and are required to define the multiplicities and ranges of compound data element types. The
range type is used to define a domain and associate its cardinality to a variable name. Thus, the declaration

\[
\langle \text{elementName} \rangle : \text{range}(\langle \text{domainName} \rangle) ;
\]

defines the domain \text{domainName} of cardinality \text{elementName}. As such, the element \text{domainName} can be used in subsequent declarations for defining compound types or value types. Moreover, \text{domainName} may be referenced as a constant within model statement expressions, where it is treated as an integer element. For example, in the Bayesian Lasso, the domains of regressors and responses would be declared with range elements \( p \) and \( n \), respectively:

```plaintext
data {
  p: range(regressors);
  n: range(responses);
};
```

An element with type value is used to reference a single member of a given domain:

\[
\langle \text{elementName} \rangle : \text{value}(\langle \text{domainName} \rangle) ;
\]

### Compound Types

Compound types allow for declaring structured elements that contain multiple sub-elements of a particular sub-type (which may, in turn, be an atomic or compound type). Compound elements are classified as three types: array, map and set.

An element with array type is a dense structure indexed by one or more dimension elements:

\[
\langle \text{elementName} \rangle : \text{array}[\langle r_1 \rangle, \langle r_2 \rangle, \ldots] \text{ of} \ (\langle \text{elementType} \rangle) ;
\]
where $r_1, r_2, \ldots$ are element names of type `range`, defining the dimensions of the array that contains an element of type `elementType` in each cell. Note that the above syntax is equivalent to

$$\text{array[}\langle r_1 \rangle\text{] of array[}\langle r_2 \rangle\text{] of } \ldots \text{ of } \langle \text{elementType} \rangle;$$

For an example of `array` types, consider the Bayesian Lasso’s regressor matrix $X$ and response vector $y$, which would be declared as follows:

```plaintext
data {
  X: array[n,p] of real;
  y: array[n] of real;
  \ldots
}
```

An element with `map` type is a sparse structure indexed by one dimension element:

$$\langle \text{elementName} \rangle : \text{map[}\langle d \rangle\text{]} \text{ of } \langle \text{elementType} \rangle;$$

where $d$ is the name of a domain. For example, in the LDA model, the structure $W_{i,j}$ counts the times word $w_j$ appears in document $d_i$. Since it is unlikely for a document to contain each and every word in the dictionary at least once, a sparse structure that maps each word to its nonzero count is appropriate:

```plaintext
data {
  n: range(documents);
  m: range(words);
  w: array[n] of map[words] of integer;
  \ldots
}
```

An element with `set` type is used to denote a subset of members from a given domain, declared with the syntax:

$$\langle \text{elementName} \rangle : \text{set[}\langle d \rangle\text{]};$$
where \( d \) is the name of a domain.

In the remainder of this section, full examples of the `data` and `var` blocks for the Bayesian Lasso and LDA models are presented. Complete descriptions of the base data sets and random variables for these models can be found in Appendix A.

**Example 5.1. Bayesian Lasso**

```plaintext
data {
  n: range(responses);
  p: range(regressors);
  lambda, priorMean, priorShape, priorScale: real;
  X: array[n,p] of real;
  y: array[n] of real;
}

var {
  sigma: real;
  Beta, Tau: array[p] of real;
}
```

**Example 5.2. Latent Dirichlet Allocation**

```plaintext
data {
  n: range(docs);
  m: range(words);
  t: range(topics);
  w: array[n] of map[words] of integer;
  alpha: array[m] of real;
  beta: array[t] of real;
}

var {
  Psi: array[t,m] of real;
  Theta: array[n,t] of real;
  Z: array[n] of map[words] of array[t] of integer;
}
```

**5.1.2 Model Statements**

Model statements are used to describe how the values of declared variable elements are computed. The most basic form of model statement is the deterministic assignment,
which binds a variable expression to a general arithmetic expression that does not involve drawing samples from a distribution function:

\[
(variableExp) \leftarrow (exp) ;
\]

Deterministic assignments are helpful when describing useful intermediate symbols for computing the parameters of a particular distribution function. For example, in the Bayesian Lasso, the variable \( A \) is used in several places as shorthand for the expression \( (X^\top X + D_\tau)^{-1} \). Thus, it makes sense to make use of the following deterministic assignment:

\[
\| A \leftarrow \text{inv}(X \times X + \text{diag}(\tau));
\]

A stochastic assignment binds a variable expression to a distribution function invocation expression:

\[
(variableExp) \sim (distributionExp) ;
\]

Stochastic assignments are used to describe variables whose values are drawn randomly from a particular distribution. For example, in the Bayesian Lasso, the vector of regression coefficients \( \beta \) follows a multivariate Gaussian distribution with mean vector \( AX^\top \tilde{y} \) and covariance matrix \( \sigma^2 A \), and would be represented with the following assignment:

\[
\| Beta \sim \text{normal}(A \times X \times Y, \sigma \times A);
\]
Variable Expressions and Blocks

The expression on the left-hand side of an assignment may be the name of a variable or a reference to a variable element using a temporary indexical variable declared within the context of a block—for example, applying an assignment expression on each row of a matrix separately. Blocks can be represented using comprehension expressions or the more user-friendly for blocks.

The general syntax of a for block is:

```plaintext
for ((rangeExp_1), (rangeExp_2), ...) {
  <modelStatement_1>
  <modelStatement_2>
  ...
}
```

where each rangeExp is a range definition that may be used to iterate over the indices in a compound element or domain, or for selecting elements from a compound that satisfy a given Boolean predicate.

Consider the vector $\tau$ in the Bayesian Lasso, where each entry $\tau_i$ is drawn from an inverse Gaussian distribution with mean $\sqrt{(\lambda^2\sigma^2)/\beta_i}$ and shape parameter $\lambda^2$, represented with the following block:

```plaintext
for (i in 1:p) {
  Tau[i] ~ invGaussian(sqrt((lambda * sigma) / Beta[i]), lambda);
}
```

The referencing expressions Tau[i] and Beta[i] make use of the temporary index variable i, which corresponds to the domain and range of both variables Tau and Beta.

Multi-dimensional array elements can be “sliced” to obtain an indexed sub-array using the wildcard symbol “:.” Consider the variable X from the Bayesian Lasso being
referenced with the temporary index variables \( i \) in \( 1:n \) and \( j \) in \( 1:p \) in any of the following ways:

- \( X[i, j] \) refers to the single value \( X_{i,j} \).
- \( X[i, :] \) refers to the array\([p]\) row \( X_i \).
- \( X[:, i] \) refers to the array\([n]\) column \( (X^\top)_j \).

### Range Definitions

When describing a block, multiple kinds of range definitions are allowed. Temporary index variables over the values of a range, which may be used for referencing over array elements, are defined with the syntax:

\[
\text{(indexName) in 1:(rangeName)}
\]

Similarly, temporary index variables over the keys present in a map element are defined as follows:

\[
\text{(indexName) in (elementName).(keyDomainName)}
\]

For example, in the LDA model, the topic count assignment \( z_{i,j} \) for document \( i \) and word \( j \), represented as a vector with \( t \) entries, is obtained by sampling from a Multinomial distribution with \( n = W_{i,j} \) trials and event probabilities defined by the vector \( p = \Theta_{i,*} \times \Psi_{*,j} \) (here, “\( \times \)” denotes item-wise multiplication). The topic count assignments for the entire corpus would be sampled as follows:

```plaintext
for (i in 1:n) {
# ...
for (j in w[i].words) {
    Z[i,j] ~ multinomial(Theta[i,:] .* Psi[,:,$j, \ w[i,j]]);
}
}
```
Notice the usage of the symbol “$” in referencing the variable \( \Psi \). In general, temporary index variables defined on range types can only be used to reference array types, while temporary index variables defined over a domain may only be used to reference map types. The switching operators “$” and “@” allow for those cases, with the following provisions:

- Given the element \( \text{myMap} \) of type \( \text{map}\{\text{dom}\} \) of integer, the reference \( \text{myMap}[\@i] \) returns the integer corresponding to the key value of \( \text{dom} \) that \( i \) represents. If the map does not have an element with the key, then a default value is returned depending on the basic type, e.g., zero for numerical types, empty strings, empty maps/sets, or arrays of zeroes.

- Given the element \( \text{myArray} \) of type \( \text{array}[k] \) of integer (where \( k \) is a range over \( \text{dom} \), the reference \( \text{myArray}[\$b] \) returns the integer corresponding to the indexical value of \( i \) that is represented by the key \( b \). The array is guaranteed to contain that value, but any comprehensions using \( \$b \) result in a map type.

**Comprehension Expressions**

Ultimately, any assignment under a for block can be represented using a comprehension syntax expression. Comprehensions are a central feature of the language and are used to describe compound types and expressions under a set of range definitions. The syntax of comprehensions [76] has been proposed as a method for manipulating ordered structures with applications in database contexts [74]. Moreover, previous theoretical work in the literature demonstrates the correspondence between the type of structural recursion used in comprehension syntax and relational algebra [77].
A comprehension is an expression of the form

\[ [e|c_1, c_2, \ldots, c_k] \]

where \( e \) is an expression and each \( c_i, i \in \{1, 2, \ldots, k\} \) is a condition. Intuitively, a comprehension can be interpreted as “the collection of all \( e \) where \( c_1, c_2, \ldots \) and \( c_k \).” In this language, comprehensions are expressed as follows:

\[
\{ \langle \text{exp} \rangle | \langle \text{rangeExp}_1 \rangle, \langle \text{rangeExp}_2 \rangle, \ldots \}
\]

where each \text{rangeExp} is a range definition. For example, the \texttt{for} block with the assignments for sampling the variable \( \tau \) in the Bayesian Lasso may also be expressed with the comprehension:

```plaintext
Tau \sim \{ \text{invGaussian}(\text{sqrt}(\text{lambda} * \text{sigma}) / \text{Beta}[i]),
              \text{lambda}
         | i \text{ in } 1:p \};
```

Similarly, the assignment of the variable \( Z \) from the LDA model may be expressed with the comprehension:

```plaintext
Z \sim \{ \text{multinomial}(\text{Theta}[i,:].*\text{Psi}[::,j],
                   \text{w}[i,j])
      | i \text{ in } 1:n, j \text{ in } \text{w}[i].\text{words} \};
```

In the remainder of this section, full examples of the \texttt{init} and model blocks for the Bayesian Lasso and LDA models are presented. Complete descriptions of the Gibbs samplers for these models can be found in Appendix A.
Example 5.3. Bayesian Lasso

```r
init {
  sigma ~ invGamma(priorScale, priorShape);
  for (i in 1:p) {
    Tau[i] ~ invGaussian(priorMean, lambda);
  }
}

Ytilde <- { Y[i] - mean(Y) | i in 1:n };
A <- inv(X’*X + diag(Tau));
S <- Yhat - X * Beta;

# update Beta
Beta ~ normal(A *’ X * Ytilde, sigma * A);

# update sigma
sigma ~ invGamma(((n-1) + p) / 2,
  (inner(S) + (Beta * diag(Tau) ’* Beta)) / 2);

# update Tau
for (i in 1:p) {
  Tau[i] ~ invGaussian(sqrt((lambda * sigma) / Beta[i]),
    lambda);
}
```

Example 5.4. Latent Dirichlet Allocation

```r
init {
  for (i in 1:t) {
    Psi[i] ~ dirichlet(alpha);
  }
  for (i in 1:n) {
    Theta[i] ~ dirichlet(beta);
  }
}
```
5.2 Overview of the BUDS Translator

At a high level, the BUDS translator takes as input a stochastic model, then executes a sequence of steps in order to produce an optimized set of schemas and relational queries for executing the model in the context of MCDB/SimSQL, as depicted in

```plaintext
# update Psi
for (k in 1:t) {
    Psi[k] ~ dirichlet(alpha + sum({ Z[i,:,k] | i in 1:n }));
}

for (i in 1:n) {
    # update Theta
    Theta[i] ~ dirichlet(beta + sum({ Z[i,j,:) | j in 1:m }));

    # update Z
    for (j in w[i].words) {
        Z[i,j] ~ multinomial(Theta[i,:].* Psi[ :,$j],
                               w[i,j]);
    }
}
```
Figure 5.1. This sequence of steps can be summarized as follows:

1. First, the model is parsed. This first pass performs syntactic analysis and produces an abstract syntax tree for each model element and a table of identifiers mapping each declared variable name to its type. The structure of the syntax tree corresponding to each model element is checked for type consistency, deducing the types of each variable reference, arithmetic expression, function parameter and output.

2. The model as a whole is checked and normalized by eliminating all blocks, spurious variables and deterministic variable assignments, simultaneously checking for circular references, which are only allowed for stochastic variables. The result is a smaller model containing the definitions and syntax tree for the stochastic variables and the base, constant data elements.

3. A graph of dependencies among the random variables in the model is constructed to determine if the stochastic model is a “standard” MCDB stochastic model or a full SimSQL Gibbs sampler. For the latter, the translator checks if the set of model elements under the init section are enough to start off the MCMC simulation chain and proceeds to generate a directed acyclic graph (DAG) with the order in which the variables must be sampled.

4. A relational schema for the base data and a set of database queries for sampling the random variables in the model is generated. To this end, the translator applies a search algorithm in the space of possible schemas and query implementations to find a combination that minimizes the overall execution cost.

The remainder of this chapter concentrates on detailed descriptions of steps (2)-(4) shown above.
5.3 Model Checking and Normalization

After the model has been parsed and the types of the expressions in the model have been inferred and checked, the translator verifies that the types of each assignment are consistent, that all temporary index variables are employed correctly and that the variables being referenced are assigned once.

5.3.1 Model Normalization

The first phase of model normalization consists of breaking up and eliminating all \texttt{for} blocks and other syntactic sugar so that, in the end, only pure comprehensions remain, in the following way:

1. For each variable-expression binding under a \texttt{for} block, create a temporary copy of the block and all its parent blocks, so that each copy contains one and only one variable-expression binding.

2. From the bottom-up, transfer the elements of each \texttt{for} block into a new comprehension, eliminating the block and the temporary index variables used in the left-hand side of the binding.

3. Each comprehension expression that is generated at a level is collapsed with the resulting parent comprehension.

To illustrate this, consider the following block from the LDA model:

```plaintext
for (i in 1:n) {
    Theta[i] ~ dirichlet(beta + sum({ Z[i,j,:] | j in 1:m }));

    for (j in w[i].words) {
        Z[i,j] ~ multinomial(Theta[i,:] .* Psi[ :, $j],
                             w[i,j]);
    }
}
```
During the first step, the outermost for block is split:

```plaintext
for (i in 1:n) {
  Theta[i] ~ dirichlet(beta + sum({ Z[i,j,:] | j in 1:m }));
}

for (i in 1:n) {
  for (j in w[i].words) {
    Z[i,j] ~ multinomial(Theta[i,:].* Psi[:,$j],
                      w[i,j]);
  }
}
```

During the second step, the transformation produces:

```plaintext
Theta ~ { dirichlet(beta + sum({ Z[i,j,:] | j in 1:m })) | i in 1:n };

for (i in 1:n) {
  Z[i] ~ { multinomial(Theta[i,:].* Psi[:,$j],
                  w[i,j]) | j in w[i].words };
}
```

At this point, Theta is finished, but the block around the assignment on Z remains. Thus:

```plaintext
Theta ~ { dirichlet(beta + sum({ Z[i,j,:] | j in 1:m })) | i in 1:n };
Z ~ { multinomial(Theta[i,:].* Psi[:,$j],
                 w[i,j]) | j in w[i].words | i in 1:n };
```

Finally, the third step merges the nested comprehension expressions in Z:

```plaintext
Theta ~ { dirichlet(beta + sum({ Z[i,j,:] | j in 1:m })) | i in 1:n };
Z ~ { multinomial(Theta[i,:].* Psi[:,$j],
                 w[i,j]) | i in 1:n, j in w[i].words };
```
After all the for blocks have been eliminated and all variables have a comprehension expression on the right-hand side, the second phase of the normalization process verifies that all deterministic variable bindings are non-circular, and then proceeds to eliminate them.

Let $G = (V, E)$ denote a variable dependency graph where the set of vertices $V$ contains all the variable elements in the model and the directed edge $(u, v) \in E$ if the defining expression of variable $v$ references variable $u$—that is, that $v$ “depends on” $u$. A deterministic variable $v_0$ is circular if $G$ contains a path $P = \langle v_0, v_1, v_2, \ldots, v_k, v_0 \rangle$ where at least one $v_i, i \in \{1, 2, \ldots, k\}$ is deterministic.

The variable dependency graph for the Bayesian Lasso is shown in Figure 5.2. Deterministic variable elements are displayed with a gray background. Notice that no circular deterministic variables are present—any paths starting and ending at $S$, $A$ and $\tilde{Y}$ contain at least one non-deterministic vertex.
After all the deterministic variables have been checked as non-circular, each one is eliminated from the model by replacing every reference to a variable with a copy of its defining expression, repeating comprehensions and indexed compound references whenever necessary.

5.3.2 Initialization and Sampling Order

Once the model has been normalized, the translator proceeds to check the dependency graph to determine what kind of stochastic model is described. Generally, the lack of cycles in the dependency graph implies that the basic MCDB capabilities are enough to run the simulation. On the other hand, the existence of a single cycle in the graph indicates that the model is an MCMC Gibbs sampler that requires the SimSQL extensions that make use of stochastic table references with generation number indices.

To start off the sampler, some of the random variables in the model must have definitions for the “zeroth” generation. These definitions are located in the init section of the model code. Therefore, the first step is to determine if the variables defined in the init section suffice to start off the sampler.

Notice that, in the model definition, random variables are referenced by name, without mention of their generation indices. This information, which is required by MCDB/SimSQL and determines the order in which the random variables must be sampled and conditioned, is inferred from the dependency graph.

Let $G = (V, E)$ denote the variable dependency graph for the model with vertices $V$ and edges $E$. Let $V_0 \subseteq V$ denote the dependency graph for the variables defined in the init section. To determine if those init variable definitions are enough to start off the chain, we check if $V_0$ forms a feedback vertex set on $G$—that is, if removing $V_0$
The stochastic variable dependency graphs for the Bayesian Lasso and LDA models are shown in Figure 5.3. Since the graph for the Lasso is a full graph, any feedback vertex set must be of size $n - 1$, where $n = |V|$. In the example specification, $V_0 = \{\text{Sigma}, \text{Tau}\}$, which is a feedback vertex set. In the LDA, the specification states that $V_0 = \{\text{Theta}, \text{Phi}\}$, which is one of the two possible feedback vertex sets according to the dependency graph, the other being $\{Z\}$.

Finding out the sampling order and the generation reference indices for the model is done by examining $G$ and the definitions in the init section to produce a chain dependency graph. Observe that, although it is possible to specify all the random variables in the init section, it is rarely necessary to use all of them during the zeroth iteration to start off the Markov chain—for instance, in the Bayesian Lasso, sampling from two out of the three random variables in the full graph is enough.

The translator makes use of a simple greedy procedure to decide which initializers to keep for the first iteration and simultaneously obtain the chain dependency graph with the annotated generation indices. The procedure is shown in Algorithm 5.1. The
**Algorithm 5.1: ChainOrder** \( G = (E, V); G_0 = (E_0, V_0) \)

```
while \( G \) has cycles do
    Let \( \hat{V} = \{ v \in V_0 \mid v \) has no incoming edges in \( G_0 \} \)
    \( \triangleright d_{in} \) and \( d_{out} \) return the in-degree and out-degree of a vertex in \( G \)
    \( v_{\text{max}} = \arg\max_{v \in \hat{V}} \left[ \min\{d_{in}(v), d_{out}(v)\} \right] \)
    \( \triangleright v^* \) represents the variable \( v_{\text{max}} \) on generation \( i + 1 \)
    \( V = V \cup \{ v^* \} \)
    \( \triangleright \) Re-route edges ending at \( v_{\text{max}} \) to \( v^* \)
    \textbf{foreach} \( (u, v_{\text{max}}) \in E \) \textbf{do}
    \begin{align*}
    E &= E \setminus \{(u, v_{\text{max}})\} \\
    E &= E \cup \{(u, v^*)\}
    \end{align*}
    \( \triangleright \) Update \( G_0 \)
    \( V_0 = V_0 \setminus \{ v_{\text{max}} \} \)
    \( E_0 = E_0 \setminus \{(u, v_{\text{max}}) \mid u \in V_0 \} \)
```

resulting graph is acyclic and makes a distinction between a variable in the *current* generation and a variable in the *next* generation by appending an asterisk symbol “*” to the label of the latter.

The chain dependency graphs that result from applying ChainOrder to the variable dependency graphs of the Bayesian Lasso and LDA model are shown in Figures 5.4(a) and 5.4(b), respectively.

### 5.4 Searching for Schemas and Query Implementations

After a model has been checked and normalized, the translator proceeds to generate a full implementation of the model that may be executed in MCDB/SimSQL. This implementation includes two elements:

1. A relational database schema that describes a set of tables for storing the base
Figure 5.4: Chain variable dependency graphs for the Bayesian Lasso (a) and LDA (b) models.

2. A set of query implementations, represented as stochastic CREATE TABLE statements, for drawing samples from the variables of the model.

5.4.1 A Naive Solution

A simple, straightforward translation of the model consists of generating a purely relational database schema for the model and, then, generating query implementations (which are stored in a library) for each operation in the model description. Consider, for example, the variable $X$ from the Bayesian Lasso, represented as the data element

$$X: \text{array}[n,p] \text{ of real};$$

To generate a schema for storing $X$, it suffices to define a table containing $n \times p$ records, each with three attributes:
• **id_response**: an INT that identifies the rows $X_1, X_2, \ldots, X_n$.

• **id_regressor**: an INT that identifies the columns $(X^\top)_1, \ldots, (X^\top)_p$.

• **value_X**: a DOUBLE with the value for $x_{i,j}$.

Then, assume the following expression is to be translated

$$\text{inv}(X \ '.* X)$$

The generation algorithm proceeds in bottom-up fashion. Thus, it generates a query implementation for the transpose-and-multiply operator ‘*’, by looking in the library of template implementations according to the representation of its operands, e.g.

```plaintext
tmul : array[m,n] \times array[m,n] \rightarrow array[m,m]
tmul : T1(id_1 INTEGER, id_2 INTEGER, val DOUBLE) \times T2(id_1 INTEGER, id_2 INTEGER, val DOUBLE) 
    \rightarrow (id_{1a} INTEGER, id_{1b} INTEGER, val DOUBLE) 
\rightarrow SELECT T1.id_1, T2.id_1, SUM(T1.val \times T2.val) 
FROM T1, T2 
WHERE T1.id_2 = T2.id_2 
GROUP BY T1.id_1, T2.id_1;
```

Resulting in the query implementation as a temporary view

```plaintext
CREATE VIEW x_tmul_x(id_regressor1, id_regressor2, value) AS 
SELECT T1.id_regressor, T2.id_regressor, 
    SUM(T1.value_X \times T2.value_X) 
FROM X as T1, X as T2 
WHERE T1.id_response = T2.id_response 
GROUP BY T1.id_regressor, T2.id_regressor;
```

At this point, the translator proceeds to generate an implementation for the matrix inverse function `inv`, following the same procedure of looking for a usable implementation according to the representation of its parameters—that is, on the schema of the view `x_tmul_x` defined above.
The solution described so far assumes a single possible relational representation for the base data set, and a single possible implementation for the operations in the model description. However, database systems such as MCDB/SimSQL have a rich type system that permit alternative representations of a given data element by allowing attributes with real-valued VECTOR or MATRIX data types in a given table. In such cases, the variable $X$ from the Bayesian Lasso may be represented with any of the following four schemas:

1. The pure relational implementation shown above, containing $n \times p$ records:

$$X(\text{id\_response INTEGER, id\_regressor INTEGER, value\_X DOUBLE})$$

, or

2. A column-vector based representation, containing $n$ records:

$$X(\text{id\_response INTEGER, value\_X VECTOR}[p])$$

, or

3. A row-vector based representation, containing $p$ records:

$$X(\text{id\_regressor INTEGER, value\_X VECTOR}[n])$$

, or

4. A pure matrix representation, containing a single record:

$$X(\text{value\_X MATRIX}[n,p])$$
Given that many possible representations are allowed, it follows that many different implementations for the same operation or function would be available—for instance, the implementation for the transpose-and-multiply operation with pure relational operands differs from the implementation with pure matrix representations. Even then, there exist a wide variety of matrix multiplication or inversion algorithms that may be used for the pure-matrix representation. Thus, one can have an entire set of template implementations for an operation, such as:

\[
\text{tmul : array}[m,n] \times \text{array}[m,n] \rightarrow \text{array}[m,m]
\]

\[
\text{tmul : T1(id}_1 \ \text{INTEGER}, \text{id}_2 \ \text{INTEGER}, \text{val DOUBLE}) \times \\
\text{T2(id}_1 \ \text{INTEGER}, \text{id}_2 \ \text{INTEGER}, \text{val DOUBLE}) \\
\rightarrow (\text{id}_1a \ \text{INTEGER, id}_1b \ \text{INTEGER, val DOUBLE}) \\
\Rightarrow \text{SELECT T1.id}_1, \text{T2.id}_1, \text{SUM(T1.val * T2.val)} \\
\text{FROM T1, T2} \\
\text{WHERE T1.id}_2 = \text{T2.id}_2 \\
\text{GROUP BY T1.id}_1, \text{T2.id}_1;
\]

\[
\text{tmul : T1(val MATRIX}[m,n]) \times \text{T2(val MATRIX}[m,n]) \\
\rightarrow (\text{val MATRIX}[m,m]) \\
\Rightarrow \text{SELECT mmul(transpose(T1.val_matrix), T2.val_matrix)} \\
\text{FROM T1, T2;}
\]

\[
\text{tmul : T1(val MATRIX}[m,n]) \times \text{T2(val MATRIX}[m,n]) \\
\rightarrow (\text{val MATRIX}[m,m]) \\
\Rightarrow \text{SELECT mtmul(T1.val_matrix, T2.val_matrix)} \\
\text{FROM T1, T2;}
\]

Taking into account the complexity of the search space, clearly the naive solution described so far is not complete. We describe a more appropriate search strategy in the remainder of this section.

### 5.4.2 Representing the search space

The full search space of the problem includes:
1. The set of possible schemas for storing the variables in the base data set, and
2. The set of possible implementations for the operations in the CREATE TABLE
   statements of the variables in the model.

To navigate the search space, we define formal structures for representing a given
solution.

**Type lattices**

Every base data variable and expression in the model is associated with a *type lattice*, which is a hierarchical structure that embodies the multiple concrete, relational
schemas of said element. Formally, a type lattice is a strongly connected graph
$L = (V, E, \hat{v})$ where

- A vertex $v \in V$ denotes a possible relational representation.
- Each edge $e \in E$ of the form $(v_a, v_b, \rho_{a\rightarrow b}, \rho_{b\rightarrow a})$ connects the representations $v_a, v_b \in V$ with the transition query templates $\rho_{a\rightarrow b}$ and $\rho_{b\rightarrow a}$.
- The vertex $\hat{v} \in V$ denotes the *current* representation of the data type.

In a type lattice, two representations $v_a$ and $v_b$ are connected with an edge if the data
element can switch from $v_a$ to $v_b$ and vice-versa using the transition queries $\rho_{a\rightarrow b}$ and $\rho_{b\rightarrow a}$, respectively. The type lattice for the variable $X$ from the Bayesian Lasso is
shown in Figure 5.5, with the current representation $\hat{v}$ marked in gray background.

To exemplify such transition queries, consider the edge that connects the purely-
relational representation $X(id_1, id_2, value)$ with the representation column-
based representation $X(id_2, value)$, containing the query $\rho_{3\rightarrow 4}$.
Figure 5.5: Type lattice for the variable $X$ from the Bayesian Lasso.

| SELECT x.id_2 AS id_2, VEC(x.value, LABEL(x.id_1)) AS value |
| FROM X as x; |
| GROUP BY x.id_2; |

where the special-purpose aggregate function VEC takes a set of numeric attribute values and creates a single VECTOR type, indexed using the attribute id_1.

Conversely, the query $\rho_{4 \rightarrow 3}$ is as follows:

| SELECT r.id_response AS id_1, x.id_2 AS id_2, GetScalar(x.value, r.id_1) AS value |
| FROM X as x, response AS r; |

Here, the table responses contains $n$ records with the indexing values of id_1, which are used by the function GetScalar to obtain the individual values from the VECTOR type.
Function signatures

The BUDS translator makes use of a library of function implementations containing templates from which queries may be generated. Each function $f$ is associated with a primary signature that uses the abstract BUDS data types and follows the form

$$f : t_1 \times t_2 \times \ldots \times t_n \rightarrow t_{\text{out}}$$

where $t_1, t_2, \ldots, t_n$ are the abstract data types of the input parameters, and $t_{\text{out}}$ is the data type of the output of the function. The BUDS language permits overloading functions based on name, such as the $*$ operator which can be used for scalar, vector and matrix multiplication; however, multiple functions with the same name and input abstract types are not supported.

In addition to the primary signature, a function is associated with multiple implementations, each containing a secondary signature that uses the concrete relational data types and follows the form

$$f : T_1 \times T_2 \times \ldots \times T_n \rightarrow T_{\text{out}} \rightarrow Q$$

where $T_1, T_2, \ldots, T_n$ are the concrete relational representations of the input parameters, $T_{\text{out}}$ is the concrete relational representation of the output of the query, and $Q$ is the query template for the implementation. The BUDS language allows multiple implementations with the same secondary signature.

Expression graphs

Each possible solution (that is, all the operations necessary for sampling each of the random variables in the model) is represented with a directed acyclic graph known as an expression graph $G = (V, E)$, in which every vertex $V$ denotes a computation
(such as a function invocation) and $E$ denotes input and referencing—thus, the edge $v_a \rightarrow v_b$, where $v_a, v_b \in V$, denotes that the operation in vertex $v_b$ takes as input the output of the operation in vertex $v_a$.

A vertex $v \in V$ may take one among different forms, depending on the kind of computation being performed, and is associated with a type lattice $v.L$ denoting the types of the output of the computation. Computations fall into the following categories:

1. Data vertices of the form $(V, L)$ where $V$ denotes a stochastic variable or variable from the base data set. Data vertices are always on the leaf level of the graph.

2. Operation vertices of the form $(f, L)$, where $f$ is a concrete secondary function signature, used to denote the application of arithmetic operators and functions.

3. Indexing vertices of the form $(I, L)$, where $I$ is a sequence of index variables $\langle i_1, i_2, \ldots, i_k \rangle$, denoting indexing references such as $X[i_1, i_2, \ldots, i_k]$.

4. Comprehension vertices of the form $(C, B, L)$, where $C$ is a sequence of index variables $\langle c_1, c_2, \ldots, c_k \rangle$ and $B$ is a set of Boolean predicates $b_1, b_2, \ldots, b_m$, denoting comprehension expressions such as $\{X | c_1, c_2, \ldots, c_k, b_1, b_2, \ldots, b_m\}$.

One of the most important properties of expression graphs is that, because of model normalization, all intermediate vertices have a single parent vertex. Thus, only data (leaf) vertices may have two or more parent vertices.

To exemplify expression graphs, consider the normalized expression from the Bayesian Lasso

$$
\text{Beta} \sim \text{normal}(\text{inv}(X' \times X + \text{diag(Tau)}) \ast X \ast y),
$$

$$
\sigma \ast \text{inv}(X' \ast X + \text{diag(Tau)});
$$
Figure 5.6: Expression graph for portions of the Bayesian Lasso.
The expression graph for this portion of the model is shown in Figure 5.6, with the data vertices marked in bold letters.

**A Datalog notation**

To provide a clear, formal description of the methods and transformation for traversing the search space, a Datalog-based notation of the elements of this space (type lattices, function signatures and expression graphs) is provided.

**Type lattices.** To represent type lattices, the following Datalog facts are employed:

- \( \text{typeLattice}(\text{tlName}, \text{curDTName}, \text{numIdx}) \) denotes a type lattice of name \( \text{tlName} \), where the “current” data type corresponds to the vertex named by \( \text{curDTName} \), and the number of dimensions or indices of the abstract type is denoted by \( \text{numIdx} \).

- \( \text{dataType}(\text{dtName}, \text{tlName}, \text{valAttName}, \text{valAttType}) \) denotes a vertex or data type of name \( \text{typeName} \), belonging to the lattice \( \text{tlName} \), with value attribute called \( \text{valAttName} \) of concrete type \( \text{valAttType} \).

- \( \text{indexAtt}(\text{attName}, \text{dtName}, \text{indexNum}) \) denotes an index attribute of name \( \text{attName} \), belonging to the data type \( \text{dtName} \), corresponding to the index number \( \text{indexNum} \) (which is always smaller than \( \text{numIdx} \)).

- \( \text{dataEdge}(\text{tlName}, \text{dtName1}, \text{dtName2}, \text{qName12}, \text{qName21}) \) denotes an edge in the type lattice of name \( \text{tlName} \), connecting the data types \( \text{dtName1} \) and \( \text{dtName2} \) with the transition queries named by \( \text{qName12} \) and \( \text{qName21} \).

Let us return to the example from Figure 5.5, which could be represented using the following set of Datalog facts:
Function signatures. Primary function signatures are represented using the following atomic formulas:

- `functionPrimary(fName, fID, numParams, outType)` denotes the function `fName`, which takes `numInputAtts` input parameters and returns a value of type `outType`. The element `fID` is a unique identifier for the primary function signature, given that functions may be overloaded based on name.

- `inputTypePrimary(fID, paramNum, paramType)` denotes that, in the function identified by `fID`, the type of the input parameter indexed by `paramNum` is `paramType`.

Consider the operator `+`, which is overloaded for scalar, vector and matrix addition. The operator may be represented with multiple primary function signatures, such as:

```
functionPrimary(add, addVector1, 2, array[p]).
functionPrimary(add, addMatrix1, 3, array[n,n]).
inputTypePrimary(addVector1, 0, array[p]).
inputTypePrimary(addVector1, 1, array[p]).
inputTypePrimary(addMatrix1, 0, array[n,n]).
inputTypePrimary(addMatrix1, 1, array[n,n]).
```
Regarding secondary function signatures, these are represented with the following atomic formulas:

- `functionSecondary(fID, implID, numOutIdx, outValAttType)` denotes that the function identified by the primary signature name `fID` is associated with the implementation from query template `implID`, which returns a relation with `numOutIdx` index attributes and a value attribute of type `outValAttType`.

- `inputParamSecondary(implID, paramNum, numIdx, valAttType)` denotes that the parameter indexed by `paramNum` in the function implementation `implID` has `numIdx` index attributes and a value attribute of type `valAttType`.

To exemplify the above, take the three different implementations for the transpose-and-multiply operator `' *` shown in Section 5.4.1, which may be represented as follows:

```
functionSecondary(tmul, tmulImpl1, 2, double).
functionSecondary(tmul, tmulImpl2, 0, matrix).
functionSecondary(tmul, tmulImpl3, 0, matrix).
inputParamSecondary(tmulImpl1, 0, 2, double).
inputParamSecondary(tmulImpl1, 1, 2, double).
inputParamSecondary(tmulImpl2, 0, 0, matrix).
inputParamSecondary(tmulImpl2, 1, 0, matrix).
inputParamSecondary(tmulImpl3, 0, 0, matrix).
inputParamSecondary(tmulImpl3, 1, 0, matrix).
```

Expression graphs. Given the multiple kinds of vertices allowed in expression graphs, the Datalog representation requires the usage of multiple types of atomic formulas, including:

- `dataV(vName, varName, tlName)` denotes a data vertex `vName`, associated with the type lattice `tlName`, and used to read the variable `varName`. 
- `operationV(vName, fID, implID, tlName)` denotes an operation vertex named `vName`, associated with the type lattice `tlName`, which invokes the function corresponding to the primary signature identifier `fID` via the secondary implementation identifier `implID`.

- `indexingV(vName, numIdx, tlName)` denotes an indexing vertex named `vName`, associated with the type lattice `tlName`, which de-references `numIdx` variables.

- `compV(vName, numIdx, numPreds, tlName)` denotes a comprehension vertex named `vName`, associated with the type lattice `tlName`, containing `numIdx` variables for comprehension and `numPreds` filtering predicates.

- `indexingVar(vName, indexNum, varName)` denotes that the index `indexNum` in vertex `vName` corresponds to the variable with name `varName`.

- `compPredicate(vName, predNum, predID)` denotes that the predicate `predNum` in `vName` corresponds to the boolean expression identified by `predID`.

- `parent(vName1, vName2)` denotes a parent-child relation between `vName1` and `vName2` in the expression graph—that is, `vName1` takes as input the output of `vName2`.

- `root(vName, varName)` denotes that the vertex `vName` is one among the multiple possible root operation vertices in the model, and that it is associated with the output variable of name `varName`.

The above atomic formulas may be used to represent the graph from Figure 5.6 as follows:
5.4.3 Exploring the search space

As shown above, every possible solution for a model is represented with an expression graph. To navigate the search space, the translator takes a given solution and applies one of the available transformations, leading to a new solution that is costed and transformed further. These transformations make local changes to the expression graph by changing the labeling of a specific vertex.

Transformation: Alternative base schemas

The first transformation makes changes to the representation of a particular base data variable. It consists of selecting one of the leaf vertices corresponding to a base data variable, examining its type lattice, and changing the current vertex \( \hat{v} \) to any of the available representations in the vertex set \( V \).

This transformation can be explained with the following Datalog rule (Here, the rule \( \text{neq}(a,b) \) is true iff \( a \neq b \), and the rule \( \text{baseVariable}(v) \) is true iff \( v \) is a base, non-stochastic variable):

\[
\text{alternateBase}(V, \text{typeLattice}(L, D, N)) :- \\
dataV(V, R, L), \\
\text{baseVariable}(R), \\
\text{typeLattice}(L, C, N), \\
dataType(D, L, _, _), \\
\text{neq}(C, D).
\]
Given a leaf vertex \( v \) with type lattice \( L \), the query `alternateBase(v, X)` returns, as solutions for \( X \), all possible `typeLattice` tuples for the representations associated with \( L \), with the exception of the current representation \( \hat{v} \). Applying the query `alternateBase(V, X)` returns the set of `typeLattice` tuples for all \( V \), for a total of

\[
k = \sum_{i=1}^{n} (|L_i.V| - 1)
\]

tuples, given vertices \( v_1, v_2, \ldots, v_n \). At a given application of the transformation, one of the \( k \) possible `typeLattice` tuples is selected to replace its existing counterpart in the database, leading to a new solution. If one the query `alternateBase(V, X)` is re-issued after the transformation has been applied, a slightly different set of \( k \) “candidate” `typeLattice` tuples results, which includes a tuple containing the previous current representation \( \hat{v} \). Thus, the transformation allows for traversing the space of all possible representations for the base data variables which contains a total of

\[
\prod_{i=1}^{n} |L_i.V|
\]

representations.

**Transformation: Alternative implementations**

The second transformation makes changes to the implementation of a particular operation in the model. It consists of selecting one of the intermediate or root `operation` vertices of the model, enumerating the possible implementations under the same primary signature of the function, and switching to an alternative one.

If the vertex being transformed is a root vertex, it means that this is a stochastic function and that it defines the representation of the stochastic variable being
sampled. Thus, the current vertex $\hat{v}$ of the type lattice of the stochastic variable is changed so as to match the output type of the alternative implementation of the function.

This transformation can be explained with the following Datalog rules (Here, the rule $\text{implLatticeDataType}(L, I, D)$ is true iff the data type $D$ corresponds to the output type of the secondary function implementation $I$ in the type lattice $L$):

\begin{verbatim}
alternateImpl(V, operationV(V, F, I, L), typeLattice(L, D, N)) :-
    operationV(V, F, C, L),
    functionSecondary(F, I, _, _),
    neq(I, C),
    typeLattice(L, _, N),
    implLatticeDataType(L, I, D).

alternateImpl(V, operationV(V, F, I, L), typeLattice(K, D, N)) :-
    root(V, R),
    dataV(_, R, K),
    operationV(V, F, C, L),
    functionSecondary(F, I, _, _),
    neq(I, C),
    typeLattice(K, _, N),
    implLatticeDataType(K, I, D).
\end{verbatim}

The second rule shown above applies to root vertices specifically and updates the type lattice of the associated stochastic variable and its vertex. Given an operation vertex $v$, the query $\text{alternateImpl}(v, X, Y)$? returns, as solutions for $X$ and $Y$, a set of pairs of $\text{operationV}$ and $\text{typeLattice}$ tuples, corresponding to the alternate implementations of the function in $v$ and the updated type lattice of the output variable data vertex. Applying the query $\text{alternateImpl}(V, X, Y)$? returns the set of possible implementations and corresponding type lattices for all $V$, containing a total of

$$k = \sum_{i=1}^{m} (g(v_i) - 1)$$

elements, where $g$ is a function that returns the number of possible implementations
that can be selected for a given operation vertex, applied on each vertex $v_1, v_2, \ldots, v_m$. At a given application of the transformation, one among the $k$ possible pairs is selected to replace the existing two operation $V$ and type lattice tuples in the database, leading to a new solution. Re-executing the query $\text{alternateImpl}(V, X, Y)$ after the transformation has been applied produces a slightly different set of $k$ candidate pairs, which includes a pair for the implementation that was being employed before the transformation. In this fashion, the transformation allows for traversing the space of all possible implementations, which contains a total of

$$\prod_{i=1}^{m} g(op_i)$$

different possible combinations. ■

5.4.4 Sketch of the Search Strategy

To efficiently explore the search space without exhausting all the possible representations and implementations, a best-first search strategy with a limited queue of solutions, a limited number of iterations, and solution memoization is applied.

The algorithm maintains a queue $Q$ with maximum size $k$, where $k$ is the number of solutions that can be generated in an iteration:

$$k = \left( \sum_{i=1}^{n} |L_i.V| - 1 \right) + \left( \sum_{i=1}^{m} g(op_i) - 1 \right)$$

At a given iteration, the algorithm removes the solution from $Q$ with the lowest cost and proceeds to apply both transformations to the solution. Each newly generated solution is checked against the memoization structure so that it may be discarded if it has already been enumerated, otherwise it is added to the memoization structure.
and placed in \( Q \)—if \( Q \) is full and the cost of the new solution is smaller than the cost of the most-expensive solution in \( Q \), then the latter is removed to make space; otherwise, the new solution is discarded.

The algorithm stops whenever (a) the maximum number of iterations has been reached, or (b) no new solutions are added to \( Q \) during the last iteration. In both cases, the solution with the lowest cost in \( Q \) is returned.

**Costing Scheme**

To determine the cost of a given solution, the BUDS translator makes use of the MCDB/SimSQL query optimizer. The MCDB/SimSQL query optimizer takes as input a database query, represented as a relational algebra expression or query plan \( P \), and returns the optimized expression \( P' \) together with an estimate of the execution cost of the plan \( c(P') \).

Since each solution is made up of multiple `CREATE TABLE` statements, —one for each stochastic variable in the model—, the BUDS translator generates a separate relational algebra expression for sampling each variable and gives it to the optimizer separately. Thus, given a solution with \( n \) stochastic variables, the total cost of the solution is defined as the sum of the individual cost of each query plan:

\[
\sum_{i=1}^{n} c(P'_i)
\]

**5.5 Experimental Evaluation**

The purpose of this section is to evaluate the language I have described in this chapter. In general, the goal is to show that the language allows for expressing Gibbs samplers for machine learning models in a succinct, readable way, and that the translator
generates relational schemas and queries that, when executed, provide acceptable performance. To this end, I compare a few popular models implemented in this language with their MCDB/SimSQL counterparts.

### 5.5.1 Models Tested

The following models were tested:

1. The Bayesian Lasso for linear regression and feature selection,

2. Latent Dirichlet Allocation (LDA) for text mining and document clustering,

and

3. A Gaussian Mixture Model (GMM) for clustering real-valued data.

The full description of the Gibbs samplers and their MCDB/SimSQL implementations can be found in Appendix A. The implementations of the Bayesian Lasso and LDA in the language are given in Section 5.1. The implementation of the GMM is shown below.

**Example 5.5.** Gaussian Mixture Model

```plaintext
model GMM {
    data {
        n: range(points);
        d: range(dims);
        k: range(clusters);
        X: array[n,d] of real;
        priorMix: array[k] of real;
        priorMean: array[d] of real;
        priorCovar, priorScale: array[d,d] of real;
        priorDegrees: integer;
    }
}```
var {
  mix: array[k] of real;
  means: array[k,d] of real;
  covars: array[k,d,d] of real;
  Z: array[n,k] of integer;
  csum: array[k] of integer;
  xsum: array[k,d] of real;
  msum, vsum: array[k,d,d] of real;
}

init {
  mix ~ dirichlet(priorMix);
  for (j in 1:k) {
    covars[j] ~ invWishart(priorScale, priorDegrees);
    means[j] ~ normal(priorMean, priorCovar);
  }
  for (i in 1:n) {
    Z[i] ~ multinomialGMM(mix, means, covars, X[i]);
  }
  for (j in 1:k) {
    csum[j] <- sum(Z[:,j]);
    msum[j] <- sum({ Z[i,j] * outer(X[i,:], -means[j,:]) | i in 1:n });
    vsum[j] <- sum({ covars[j] | i in 1:n });
    xsum[j] <- sum({ Z[i,j] * X[i,:] | i in 1:n });
    covars[j] ~ invWishartGMM(priorDegrees, priorScale, csum[j], msum[j]);
    means[j] ~ normalGMM(priorMean, priorCovar, csum[j], vsum[j], xsum[j]);
  }
  mix ~ dirichlet(priorMix + csum);
}

Observe that the above implementation uses the multinomialGMM, invWishartGMM and normalGMM functions, which correspond to special-purpose VG functions that, internally, compute the actual parameters of the distributions as shown in the Gibbs sampler for the GMM. It is worth noting that the language has the capability to denote these computations directly and use the invWishart, normal and multinomial functions. However, to make a fair experimental comparison with the implementation
shown in Appendix A (which is the original implementation from [13] and [107]), the special-purpose functions are kept.

5.5.2 Setup

To test the models described above, I implemented a proof-of-concept translator for the language that includes a parser, translator and optimizer that takes as input a model description and some basic statistics regarding the domain sizes specified, and produces SQL code for creating the base data schemas and generating the random variables in the Gibbs sampler.

To compare the performance of the models generated by the translator with their hand-written SQL counterparts from Appendix A, the MCDB/SimSQL runtime was used to execute each model separately in an Amazon EC2 cluster of five m2.4xlarge machines with the following base data:

1. For the Bayesian Lasso, a synthetic data set with 100 regressor dimensions and 100,000 data points per machine.

2. For the LDA model, a data set based on combinations of the “20 newsgroups” data set, with a dictionary size of 10,000 and 250,000 documents per machine, categorized into \( k = 20 \) topics.

3. For the GMM model, a synthetic data set consisting of 1,000,000 data points per machine, each of them containing 10 dimensions, categorized into \( k = 10 \) clusters.

For each model, the wall-clock running time of the initialization (“zeroth”) iteration and the average of the first five regular iterations of the Gibbs sampler were collected.
Table 5.1: Number of lines of code and running times (MM:SS) for hand-written SQL and language implementations of the Lasso, LDA and GMM models.

<table>
<thead>
<tr>
<th>Model</th>
<th>Lines of code</th>
<th>Running time (MM:SS)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SQL</td>
<td>BUDS</td>
</tr>
<tr>
<td>Lasso</td>
<td>119</td>
<td>32</td>
</tr>
<tr>
<td>LDA</td>
<td>84</td>
<td>33</td>
</tr>
<tr>
<td>GMM</td>
<td>129</td>
<td>40</td>
</tr>
</tbody>
</table>

Regarding ease of implementation, I compared the number of lines of code (LOC) required to implement each model in the language vis-a-vis the SQL implementations from Appendix A. Comments and blank lines are not counted for either language and, in the case of SQL, the length of the CREATE TABLE statements for the base data tables is included.

5.5.3 Results and Discussion

Table 5.1 shows the results of the experiments, together with a comparison of the number of lines of code required for each implementation. The times in parentheses represent the time for executing the zeroth iteration.

In general, models written in the language described in this chapter require a shorter amount of code than those in SQL. Furthermore, the code written in the language is easier to understand and reflects the mathematical descriptions of the models better than the code written in SQL.

Regarding performance, the results for all three models show that there is an acceptable amount of overhead in execution time for the generated code that the compiler for the language produces, compared to the hand-written SQL code. This
overhead is a result of the way the translator generates sub-queries for the operations
corresponding to the model expressions which, although semantically equivalent to the
hand-written SQL code, cannot be optimized further by the MCDB/SimSQL query
optimizer. To illustrate this, let us compare the hand-written code for the Bayesian
Lasso, shown in Appendix A.1, with the code generated by the BUDS translator.
Consider the following computation in the sampling of the random variable $\sigma^2$:

$$(\tilde{y} - X\beta)^\top(\tilde{y} - X\beta) + (\beta^\top D_{\tau} \beta)$$

Assuming a purely relational representation of the variables $\tilde{y}$, X, $\beta$ and $\tau$, the
hand-written SQL computation would comprise several nested sub-queries:

1. First, compute the matrix-vector multiplication $X\beta$ using a SUM aggregate for
   the expression
   $$\sum_{j=1}^{p} x_{i,j} \times \beta_j$$
   grouped by each response $i = 1, 2, \ldots, n$. Let $w$ denote the result of this
   computation, stored as a table with $n$ records.

2. Compute the sum of squares $(\tilde{y} - X\beta)^\top(\tilde{y} - X\beta)$ using a SUM aggregate for
   the expression
   $$\sum_{i=1}^{n} (\tilde{y}_i - w_i)^2$$
   Let $s_b$ denote the result of this computation, stored as a table with a single
   record.
3. Compute the multiplication $\beta^T D \tau \beta$ using a SUM aggregate for the expression

$$\sum_{j=1}^{p} \beta_j^2 \times \frac{1}{\tau_j}$$

Let $s_t$ denote the result of this computation, stored as a table with a single record.

4. Compute the final result $s_b + s_t$.

To compare the computation shown above with the one corresponding to the SQL code generated by the BUDS translator, let us examine the corresponding BUDS expression

$$\text{inner}(Yhat - X \star \text{Beta}) + (\text{Beta }^\prime \times \text{diag}(\text{Tau}) \times \text{Beta})$$

where, given a vector $y$ with $k$ entries, the inner function computes the inner product $y^T y$, and the diag function returns a $k \times k$ matrix diagonalized with the values of $y$ and zeroes everywhere else. This way, the translator generates SQL code for the following series of computations:

1. First, compute the matrix-vector multiplication $X \beta$, corresponding to the expression $X \star \text{Beta}$, using a SUM aggregate for the expression

$$\sum_{j=1}^{p} x_{i,j} \times \beta_j$$

grouped by each response $i = 1, 2, \ldots, n$. Let $w$ denote the result of this computation, stored as a table with $n$ records.

2. Compute the vector $z = y - w$ using a relational join and the scalar arithmetic computation $y_i - w_i$, storing the result as a table with $n$ records.
3. For the inner function, compute the expression $\mathbf{z}^\top \mathbf{z}$ using a SUM aggregate of the form

$$\sum_{i=1}^{n} z_i^2$$

producing the table $s_b$ with a single record.

4. For the diag function, compute a relational cross product of the table storing the variable $\tau$, and apply a SQL CASE statement for setting the diagonal elements of the matrix. The result is a table $\mathbf{D}$ with $p^2$ records.

5. Compute the matrix-vector multiplication $\mathbf{\beta}^\top \mathbf{D}$ using a relational join and the SUM aggregate

$$\sum_{j=1}^{p} \beta_j \times D_{j,k}$$

grouped by each regressor $k = 1, 2, \ldots, p$. The result is a table $\mathbf{B}$ with $p$ records.

6. Compute the expression $\mathbf{B}\mathbf{\beta}$ using a relational join and the SUM aggregate

$$\sum_{j=1}^{p} B_j \times \beta_j$$

producing the table $s_t$ with a single record.

7. Compute the final result $s_b + s_t$.

Comparing the above set of computations with the ones corresponding to the hand-written SQL code, it becomes clear that the overhead has multiple sources. First, in the hand-written SQL code, the computation corresponding to the diag function has been abstracted away from and integrated into the SUM aggregate that computes the matrix-vector multiplications—it is hard for a query optimizer to recognize that these two computations are semantically equivalent. The other source
of overhead comes from the fact that the SQL code generated by the BUDS translator computes the expression \((\tilde{y} - X\beta)^\top(\tilde{y} - X\beta)\) using an additional step prior to the aggregation—in this case, additional transformations can be added to the MCD-B/SimSQL optimizer for handling similar cases.
Chapter 6

Conclusions

In this work, I have studied three different problems in query processing and query optimization for relational databases applied to stochastic analytics. Working in the context of the MCDB/SimSQL system for Monte Carlo sampling-based analytics, I presented solutions for each of the three problems and validated them empirically.

The first part is concerned with evaluating queries in MCDB/SimSQL in a way that provides probabilistic guarantees over its query result sets. I proposed the usage of so-called probability threshold queries, described techniques that extend the database system to evaluate such queries efficiently, and executed an empirical study of these techniques. The study shows that the proposed techniques make judicious use of system resources, especially when they work in tandem with the End-Biased hypothesis test.

The second part addresses the problem of caching the intermediate result sets of a query for re-use in subsequent queries. What distinguishes the techniques I introduced from conventional query caching and tuning frameworks is the usage of a history-based cost model in the query optimizer that evaluates the potential benefits of executing sub-optimal query plans that create useful intermediate result sets. The empirical evaluation shows a significant improvement in workload running time, compared to previously-existing query caching solutions.

The third and last part is concerned with the design of a language and notation for describing stochastic models in a natural and succinct way, in contrast with the
MCDB/SimSQL extended SQL interface. I described a notation and language that allows for specifying models using structured types, such as arrays and sets, in which random variables are first-class objects, with operations based on linear algebra, relational databases and statistics. I presented techniques for translating models into optimized relational schemas and MCDB/SimSQL query implementations. Results show that models can be specified in a short and natural fashion and, when executed in MCDB/SimSQL, provide performance comparable with a hand-written and optimized SQL implementation.
Appendix: Machine Learning Models in MCDB/SimSQL

A.1 The Bayesian Lasso

The data set for the Bayesian Lasso [18] contains two elements: a regressor matrix $X$ with $n$ rows and $p$ columns, and a response vector $y$ of length $n$. The Gibbs sampler defines sampling distributions for updating the random variables $\beta_1, \beta_2, \ldots, \beta_p$ corresponding to the regression coefficients, alongside the feature variables $\tau_1, \tau_2, \ldots, \tau_p$ and the variance $\sigma^2$.

Given $\tilde{y} = y - \mu_y$, $D_\tau = \text{diag}(\tau_1, \tau_2, \ldots, \tau_p)$, $A = X^\top X + D_\tau$, and the Lasso parameter $\lambda$, the sampling distributions are:

$$
\begin{align*}
\beta &\sim \text{MultivariateNormal}\left((A^{-1}X^\top \tilde{y}, \sigma^2 A^{-1})\right) \\
\sigma^2 &\sim \text{InverseGamma}\left(\frac{(n-1)+p}{2}, \frac{(\tilde{y} - X\beta)^\top (\tilde{y} - X\beta) + (\beta^\top D_\tau \beta)}{2}\right) \\
\tau_j &\sim \text{InverseGaussian}\left(\frac{\lambda^2 \sigma^2}{\beta_j^2}, \lambda^2\right)
\end{align*}
$$

Implementing the above Gibbs sampler in MCDB/SimSQL requires definitions for the stochastic tables $\text{sigma}$, $\text{tau}$ and $\text{beta}$ for the initialization and $i^{th}$ iteration definitions.

In a purely relational implementation, the data set is represented as three separate tables:
- regressor(respID, regID, regValue) which contains the values of \( X \), with one record per entry in the matrix, so that respID identifying the row and regID identifies the column of each regValue.

- response(respID, respValue) which contains the values of \( y \), with one record per entry in the vector identified by respID.

- prior(sigmaShape, sigmaScale, tauMean, tauScale, lambda) containing a single record with constant hyper-parameter values.

First off, a few materialized views are created to avoid redundant work on each stage of the Gibbs sampler. The view centeredResponse(respID, respValue) contains the values of \( \tilde{y} \) computed from \( y \):

```sql
CREATE VIEW centeredResponse(respID, respValue) AS
SELECT r1.respID, (r1.respValue - m.meanRespValue)
FROM response AS r1,
(SELECT AVG(r2.respValue) AS meanRespValue
FROM response r2) AS m;
```

Then, the view regressorGram(dim_i, dim_j, dim_val) contains the values of the Gramian matrix \( X^\top X \) (note that only the lower triangle is stored, as the matrix is symmetric):

```sql
CREATE VIEW regressorGram(dim_i, dim_j, dim_val) AS
SELECT r1.regID, r2.regID,
SUM(r1.regValue * r2.regValue)
FROM regressor AS r1, AS regressor r2
WHERE r1.respID = r2.respID
AND r1.regID <= r2.regID
GROUP BY r1.regID, r2.regID;
```

Finally, the view regressorSum(regID, sumValue) stores the result of the matrix-vector multiplication \( X^\top \tilde{y} \):
CREATE VIEW regressorSum(regID, sumValue) AS
SELECT reg1.regID, SUM(reg1.regValue * res1.respValue)
FROM regressor AS reg1, centeredResponse AS res1
WHERE reg1.respID = res1.respID
GROUP BY reg1.regID;

The initialization ("zeroth") iteration contains definitions for sigma and tau only, as beta can be computed from the definition for the \(i\)th iteration. These random variables are initialized using the values from the prior table as hyper-parameters:

CREATE TABLE sigma[0](sigmaValue) AS
WITH g AS InvGamma(
    SELECT p.sigmaShape, p.sigmaScale
    FROM prior AS p
)
SELECT g.outValue
FROM g;

CREATE TABLE tau[0](regID, tauValue) AS
FOR EACH r IN (SELECT DISTINCT regID FROM regressors)
    WITH ig AS InvGaussian(
        SELECT p.tauMean, p.tauShape
        FROM prior AS p
    )
SELECT r.grID, 1.0 / ig.outValue
FROM ig;
Then, the definition of \( \beta \) for all iterations, including \( i = 0 \), makes use of the temporary stochastic table \( A \) to calculate a matrix inverse with the table function \( \text{MatrixInverse} \):

```sql
CREATE TABLE A[i](regID_i, regID_j, AValue) AS
WITH mi AS MatrixInverse(
    SELECT xtx.dim_i, xtx.dim_j,
    CASE
        WHEN xtx.dim_i = xtx.dim_j
        THEN xtx.dim_val + (1.0/t.tauValue)
        ELSE xtx.dim_val
    END
    FROM tau[i] AS t, regressorGram AS xtx
    WHERE t.regID = xtx.dim_i
)
SELECT mi.outID_i, mi.outID_j, mi.outValue
FROM mi;

CREATE TABLE beta[i](regID, betaValue) AS
WITH mvn AS MultiNormal(
    (SELECT a1.regID_j, SUM(a1.AValue * rs.sumValue)
    FROM A[i] AS a1, regressorSum AS rs
    WHERE a1.regID_i = rs.regID
    GROUP BY a1.regID_j
),

    (SELECT a2.regID_i, a2.regID_j,
    a2.AValue * s2.sigmaValue
    FROM A[i] AS a2, sigma[i] AS s2)
)
SELECT mvn.outID, mvn.outValue
FROM mvn;
```
To compute \( \sigma \), the sums of squares \((\bar{y} - X\beta)^T(\bar{y} - X\beta)\) and \((\beta^T D \tau \beta)\) are calculated using aggregate sub-queries to parameterize the InvGamma VG function:

```sql
CREATE TABLE sigma[i](sigmaValue) AS
WITH g AS InvGamma(
    (SELECT (p1.numResponses - 1 + p2.numRegressors)/2.0
    FROM (SELECT COUNT(*)
        FROM responses) AS p1,
        (SELECT COUNT(DISTINCT regID)
        FROM regressors) AS p2),

    (SELECT sb.sumB + st.sumT
    FROM (SELECT SUM(((res1.respValue - xb.sumVal) * (res1.respValue - xb.sumVal)) / 2) AS sumB
    FROM centeredResponse AS res1,
    (SELECT regl.respID as respID,
        SUM(regl.regValue * b1.betaValue) AS sumVal
    FROM regressor AS regl, beta[i-1] AS b1
    WHERE regl.regID = b1.regID
    GROUP BY regl.respID) AS xb

    WHERE res1.respID = xb.respID) AS sb,

    (SELECT SUM(((b2.betaValue * b2.betaValue * (1.0 / t1.tauValue)) / 2.0) AS sumT
    FROM tau[i-1] AS t1, beta[i-1] AS b2
    WHERE t1.regID = b2.regID) as st
    )
)

SELECT g.outValue
FROM g;
```
Finally, updating \( \tau \) is straightforward:

```sql
CREATE TABLE \( \tau[i] \) (regID, tauValue) AS
FOR EACH \( r \) IN regressorIDs
    WITH ig AS InvGaussian(
        (SELECT sqrt((pr1.lambda * pr1.lambda * s.sigmaValue) / (b.betaValue * b.betaValue))
        FROM prior AS pr1, sigma[i] AS s, beta[i-1] AS b
        WHERE b.regID = r.grID
        ),
        (SELECT pr2.lambda * pr2.lambda
        FROM prior AS pr2)
    )
SELECT r.grID, (1.0 / ig.outValue)
FROM ig;
```

### A.2 Latent Dirichlet Allocation

The data set for the LDA model [108] consists of a dictionary of words \( w_1, w_2, \ldots, w_m \) and a corpus of documents \( d_1, d_2, \ldots, d_n \), represented together as a multi-dimensional structure \( W \) where \( W_{i,j} \) is a positive integer denoting the number of times that word \( w_j \) appears in document \( d_i \). Given \( k \) topics for clustering documents and words, the Gibbs sampler defines sampling distributions for random variables corresponding to the topic distribution of each document \( \Theta_i, i \in \{1, 2, \ldots, n\} \), the word distribution of each topic \( \Psi_t, t \in \{1, 2, \ldots, k\} \), and the topic count assignment for each document and word \( z_{i,j}, i \in \{1, 2, \ldots, n\}, j \in \{1, 2, \ldots, m\} \). Given prior hyper-parameter vectors \( \alpha \) of length \( m \) and \( \beta \) of length \( k \), the sampling distributions are (here, “+” and “\( \times \)”)
denotes vector addition and item-wise vector multiplication, respectively):

\[
\Psi_t \sim \text{Dirichlet}(\alpha + \sum_{i=1}^{n} z_{i,*} t)
\]

\[
\Theta_i \sim \text{Dirichlet}(\beta + \sum_{j=1}^{m} z_{i,j,*})
\]

\[
z_{i,j} \sim \text{Multinomial}(\Theta_{i,*} \times \Psi_{*,j}, W_{i,j})
\]

Implementing the above Gibbs sampler in MCDB/SimSQL requires definitions for the stochastic tables \(\psi, \theta\) and \(z\) for the initialization and \(i^{th}\) iteration definitions.

In a purely relational implementation, the data set is represented with the following tables:

- \(\text{word}(\text{wordID, alphaValue})\) containing \(m\) records, each with the corresponding value of \(\alpha\) for the word identified by \(\text{wordID}\)

- \(\text{document}(\text{docID})\) containing \(n\) records identified by \(\text{docID}\)

- \(\text{wordInDoc}(\text{docID, wordID, countNum})\) with the contents of \(W\) for each document-word pair identified by \((\text{docID, wordID})\)

- \(\text{topic}(\text{topicID, betaValue})\) containing \(k\) records, each with the corresponding value of \(\beta\) for the topic identified by \(\text{topicID}\).

The initialization (“zeroth”) iteration contains definitions for \(\psi\) and \(\theta\) only, since \(z\) can be computed from the definition for the \(i^{th}\) iteration. First, \(\psi\) is initialized using the prior values of \(\beta\) stored in the \(\text{word}\) relation:
CREATE TABLE psi[0](topicID, wordID, psiValue) AS
FOR EACH t IN topic
    WITH dir AS Dirichlet(
        SELECT w.wordID, w.alphaValue
        FROM word w
    )
    SELECT dir.outID, dir.outValue
FROM dir;

Similarly, theta is initialized using the prior values of $\alpha$ stored in the topic relation:

CREATE TABLE theta[0](docID, topicID, thetaValue) AS
FOR EACH d IN document
    WITH dir AS Dirichlet(
        SELECT t.topicID, t.betaValue
        FROM topic t
    )
    SELECT dir.outID, dir.outValue
FROM dir;

Then, the definition of $Z$ for all iterations, using a Multinomial distribution to sample the topic counts:

CREATE TABLE z[i](docID, wordID, topicID, zValue) AS
FOR EACH dw IN wordInDoc
    WITH mn AS Multinomial(
        (SELECT t.topicID, p.psiValue * t.thetaValue
        FROM theta[i] AS t, psi[i] AS p
        WHERE t.docID = dw.docID
        AND p.wordID = dw.wordID
        AND t.topicID = p.topicID),
        (SELECT dw.countNum)
    )
    SELECT dw.docID, dw.wordID, mn.outID, mn.outValue
FROM mn;

The definitions of psi and theta for all iterations require the use of group-by aggregate sub-queries to obtain the parameters of the Dirichlet distributions:
CREATE TABLE psi[i](topicID, wordID, psiValue) AS
FOR EACH t IN topic
  WITH dir AS Dirichlet(
    SELECT w.wordID, w.alphaValue + ss.sumValue
    FROM word AS w,
      (SELECT z1.wordID AS wordID,
          SUM(z1.zValue) AS sumValue
      FROM z[i-1] AS z1
      GROUP BY z1.wordID) AS ss
    WHERE w.wordID = ss.wordID
  )
  SELECT dir.outID, dir.outValue
  FROM dir;

CREATE TABLE theta[i](docID, topicID, thetaValue) AS
FOR EACH d IN document
  WITH dir AS Dirichlet(
    SELECT t.topicID, t.betaValue + ss.sumValue
    FROM topic AS t,
      (SELECT z1.topicID AS topicID,
          SUM(z1.zValue) AS sumValue
      FROM z[i-1] AS z1
      GROUP BY z1.topicID) AS ss
    WHERE t.topicID = ss.topicID
  )
  SELECT dir.outID, dir.outValue
  FROM dir;

A.3 Gaussian Mixture Model

The Gaussian Mixture Model (GMM) is a popular method for clustering real-valued data. A GMM views the data set as being generated by a set of $k$ multivariate ($d$-dimensional) Normal distributions mixed together in a PDF of the form

$$p(x|\pi, \mu_1, \mu_2, \ldots, \mu_k, \Sigma_1, \Sigma_2, \ldots, \Sigma_k) = \sum_{i=1}^{k} \pi_i \times \text{Normal}(x|\mu_i, \Sigma_i)$$
where \( \mu_i \) and \( \Sigma_i \) denote the mean vector and covariance matrix of the \( i \)th Gaussian distribution, and \( \pi \) is a vector of mixing proportions satisfying \( \pi_i \geq 0 \) and
\[
\sum_{i=1}^{k} \pi_i = 1
\]
The Bayesian GMM described here assumes a Dirichlet(\( \alpha \)) prior on the mixing proportions \( \pi \), a Normal(\( \mu_0, \Phi^{-1} \)) prior on each mean vector \( \mu_i \), and an InverseWishart(\( \nu, \Psi \)) prior on each covariance matrix \( \Sigma_i, i \in \{1, 2, \ldots, k\} \). Furthermore, the model assumes that each data point \( x_j, j \in \{1, 2, \ldots, n\} \) gets its cluster assignment \( c_j \) from a Multinomial distribution with \( m = 1 \) trials and probability vector \( \mathbf{p}_j \) of length \( k \) where
\[
p_{j,i} \propto \pi_i \times \text{Normal}(x_j | \mu_i, \Sigma_i)
\]
The result is the following Gibbs sampler:

\[
\pi \sim \text{Dirichlet} \left( \alpha + \sum_{j=1}^{n} c_j \right)
\]
\[
\mu_i \sim \text{Normal} \left( \left( \Phi \mu_0 + n_i \Sigma_i^{-1} \right)^{-1} \sum_{j=1}^{n} c_{j,i} \times x_j, \left( \Phi \mu_0 + n_i \Sigma_i^{-1} \right)^{-1} \right)
\]
\[
\Sigma_i \sim \text{InverseWishart} \left( n_i + \nu, \Psi + \sum_{j=1}^{n} c_{j,i} \times (x_j - \mu_i)(x_j - \mu_i)^\top \right)
\]
\[
c_j \sim \text{Multinomial}(\mathbf{p}_j, 1)
\]
Implementing this Gibbs sampler in MCDB/SimSQL requires definitions for the sto-
chastic tables $\pi$, mean, covar and $c$ for the initialization and $i^{th}$ iteration definitions.

In a purely relational implementation, the data set comprises the following tables:

- $\text{cluster}$(clusterID, alphaValue) with $k$ records, each containing the prior value of $\alpha$ for the cluster identified by clusterID
- $\text{priorMean}$(dimID, meanValue) with $d$ records, each containing the prior value of $\mu_0$ for the dimension identified by dimID
- $\text{priorCovar}$(dimID_i, dimID_j, phiValue) with $d \times d$ records, each containing the prior value of $\Phi$ for the entry identified by the (dimID_i, dimID_j) pair
- $\text{priorScale}$(dimID_i, dimID_j, psiValue) with $d \times d$ records, each containing the prior value of $\Psi$ for the entry identified by the (dimID_i, dimID_j) pair
- $\text{prior}$(nuValue) containing a single record with the value of $\nu$
- $\text{data}$(dataID, dimID, dataValue) with $n \times d$ records, each containing the value of $x$ for the point and dimension identified by the (dataID, dimID) pair.

The initialization (”zeroth”) iteration contains definitions for $\pi$, mean and covar, while $c$ can be computed from the definition for the $i^{th}$ iteration. First, $\pi$ is initialized with the prior values of $\alpha$ stored in the cluster table:
CREATE TABLE pi[0](clusterID, piValue) AS 
  WITH dir AS Dirichlet(
    SELECT c.clusterID, c.alphaValue
    FROM cluster AS c
  )
  SELECT dir.outID, dir.outValue
  FROM dir;

The initializations for mean and covar are straightforward:

CREATE TABLE mean[0](clusterID, dimID, meanValue) AS
FOR EACH c IN cluster
  WITH mvn AS MultiNormal(
    (SELECT pm.dimID, pm.meanValue
     FROM priorMean AS pm),
    (SELECT pc.dimID_i, pc.dimID_j, pc.phiValue
     FROM priorCovar AS pc)
  )
  SELECT mvn.outID, mvn.outValue
  FROM mvn;

CREATE TABLE covar[0](clusterID, dimID_i, dimID_j, covarValue) AS
FOR EACH c IN cluster
  WITH iw AS InverseWishart(
    (SELECT p.nuValue
     FROM prior AS p),
    (SELECT ps.dimID_i, ps.dimID_j, ps.psiValue
     FROM priorScale AS p)
  )
  SELECT iw.outID_i, iw.outID_j, iw.outValue
  FROM iw;
To update the cluster assignments in \( c \), the special-purpose \texttt{MultinomialGMM} function takes all the parameters in the model, computes \( p \) for each data point in \( \texttt{data} \) by evaluating the Normal PDF on each data point, weighed by the mixing proportions:

```sql
CREATE TABLE \( c[i] \)(dataID, clusterID, countVal) AS
FOR EACH \( d1 \) IN (SELECT DISTINCT(dataID) AS dataID FROM data)
    WITH mng AS MultinomialGMM(
        (SELECT p.clusterID, p.piValue
            FROM pi[i] AS p),
        (SELECT m.clusterID, m.meanID, m.meanValue
            FROM mean[i] AS m),
        (SELECT c.clusterID, c.dimID_i, c.dimID_j, c.covarValue
            FROM covar[i] AS c),
        (SELECT d2.dimID, d2.dataValue
            FROM data AS d2
            WHERE d2.dataID = d1.dataID)
    )

SELECT d1.dataID, mng.outID, mng.outValue
FROM mng;
```
To update the covariance matrices, the `InverseWishartGMM` function takes the parameters $\nu, \Psi, n_i$ and the sums $\sum_j c_{j,i} \times (x_j - \mu_i)(x_j - \mu_i)^\top$:

```
CREATE TABLE covar[i] (clusterID, dimID_i, dimID_j, covarValue) AS
    FOR EACH c IN cluster
        WITH iwg AS InverseWishartGMM(
            (SELECT p.nuValue
                FROM    prior AS p),
            (SELECT ps.dimID_i, ps.dimID_j, ps.psiValue
                FROM    priorScale AS ps),
            (SELECT SUM(cc.countVal)
                FROM    c[i-1] AS cc
                WHERE cc.clusterID = c.clusterID),
            (SELECT d1.dimID_i, d2.dimID_j,
                SUM(cc.countVal * 
                    (d1.dataValue-m1.meanValue)*(d2.dataValue-m2.meanValue))
                FROM    data AS d1, data AS d2,
                        mean[i-1] AS m1, mean[i-1] AS m2
                        c[i-1] AS cc
                WHERE d1.dataID = d2.dataID
                AND d1.dataID = cc.dataID
                AND m1.dimID = d1.dimID
                AND m2.dimID = d2.dimID
                AND d2.dataID = cc.dataID
                AND cc.clusterID = m1.clusterID
                AND cc.clusterID = m2.clusterID
                AND cc.clusterID = c.clusterID)
        )
        SELECT c.clusterID, iwg.outDim_i, iwg.outDim_j, iwg.outValue
        FROM    iwg;
```
To update the mean vectors, the `NormalGMM` function takes the parameters $\mu_0, \Phi, n_i, \Sigma_i$ and the sum $\sum_j c_{j,i} \times x_j$:

```sql
CREATE TABLE mean[i](clusterID, dimID, meanValue) AS
FOR EACH c IN cluster
    WITH ng AS NormalGMM(
        (SELECT pm.dimID, pm.meanValue
            FROM priorMean AS pm),
        (SELECT pc.dimID_i, pc.dimID_j, pc.covarValue
            FROM priorCovar AS pc),
        (SELECT SUM(cc.countVal)
            FROM c[i-1] AS cc
            WHERE cc.clusterID = c.clusterID),
        (SELECT cv.dimID_i, cv.dimID_j, cv.covarValue
            FROM covar[i-1] AS cv
            WHERE cv.clusterID = c.clusterID),
        (SELECT d2.dimID, SUM(cc.countVal * d2.dataValue)
            FROM c[i-1] AS cc, data AS d2
            WHERE cc.dataID = d2.dataID
            AND cc.clusterID = c.clusterID
            GROUP BY d2.dimID)
    )

Finally, updating the mixing proportions is straightforward:

```sql
CREATE TABLE pi[i](clusterID, piValue) AS
WITH dir AS Dirichlet(
    SELECT c.clusterID, c.alphaValue + s.sumV
    FROM cluster AS c,
    (SELECT cc.clusterID,
        SUM(cc.countVal) AS sumV
        FROM c[i-1] AS cc
        GROUP BY cc.clusterID) AS s
    WHERE c.clusterID = s.clusterID
)

SELECT dir.outID, dir.outValue
FROM dir;
Bibliography


[38] Y. Kotidis and N. Roussopoulos, “DynaMat: A dynamic view management


[52] N. I. Hachem, K. Qiu, M. Gennert, and M. O. Ward, “Managing derived data in
the Gaea scientific DBMS,” in *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB)*, vol. 93, pp. 1–12, VLDB Endowment, 1993.


