BayesDB: Querying the Probable Implications of Tabular Data

by

Jay Baxter

Submitted to the Department of Electrical Engineering and Computer Science on May 23, 2014, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

BayesDB, a Bayesian database table, lets users query the probable implications of their tabular data as easily as an SQL database lets them query the data itself. Using the built-in Bayesian Query Language (BQL), users with little statistics knowledge can solve basic data science problems, such as detecting predictive relationships between variables, inferring missing values, simulating probable observations, and identifying statistically similar database entries. BayesDB is suitable for analyzing complex, heterogeneous data tables with no preprocessing or parameter adjustment required. This generality rests on the model independence provided by BQL, analogous to the physical data independence provided by the relational model. SQL enables data filtering and aggregation tasks to be described independently of the physical layout of data in memory and on disk. Non-experts rely on generic indexing strategies for good-enough performance, while experts customize schemas and indices for performance-sensitive applications. Analogously, BQL enables analysis tasks to be described independently of the models used to solve them. Non-statisticians can rely on a general-purpose modeling method called CrossCat to build models that are good enough for a broad class of applications, while experts can customize the schemas and models when needed. This thesis defines BQL, describes an implementation of BayesDB, quantitatively characterizes its scalability and performance, and illustrates its efficacy on real-world data analysis problems in the areas of healthcare economics, statistical survey data analysis, web analytics, and predictive policing.

Thesis Supervisor: Vikash Mansinghka
Title: Research Scientist
Acknowledgments

This thesis is based on collaborative research with Vikash Mansinghka, Dan Lovell, Baxter Eaves, and Patrick Shafto. I owe a great deal of thanks to all of them: my thesis advisor, Vikash Mansinghka, who has provided countless helpful insights and tips along the way, Dan Lovell for his work as lead developer on the CrossCat backend, Baxter Eaves for running experiments and working on the CrossCat backend, and Patrick Shafto for his general help, especially with usability design as BayesDB’s first user. I would also like to thank Seth Berg, Mason DeCamillis, and Ben Mayne for helping out by using BayesDB to analyze real datasets and adding features to the codebase. Additionally, thanks to Tejas Kulkarni and Ardavan Saeedi for many helpful conversations about BayesDB, and my other friends and family for always being so supportive of me. And finally, thank you to the DARPA XDATA and PPAML programs and Google for sponsoring BayesDB development and my research assistantships.
Contents

1 Introduction .................................................. 17
   1.1 Overview ................................................. 17
   1.2 Motivation .............................................. 18
   1.3 Example data analysis ................................. 20
   1.4 How does BayesDB work? .............................. 25
   1.5 Outline ................................................ 25

2 Bayesian Query Language (BQL) ........................... 27
   2.1 Completing statistical tasks with BQL ................. 27
      2.1.1 Detect predictive relationships between variables and quantify their strength 27
      2.1.2 Identify contextually similar database entries ............................... 28
      2.1.3 Infer missing values with appropriate error bars .......................... 28
      2.1.4 Generate hypothetical rows conditioned on arbitrary predicates .......... 28
   2.2 Formal Language Description .......................... 29
      2.2.1 Loading Data ....................................... 29
      2.2.2 Initializing Models and Running Analysis ..................................... 29
      2.2.3 Btable Administration .................................. 30
      2.2.4 Saving and Loading Btables and Models ............................... 30
      2.2.5 Querying .......................................... 31
      2.2.6 Query Modifiers ...................................... 32
      2.2.7 Column Lists ........................................ 32
      2.2.8 Predictive Functions .................................. 33
   2.3 Model Independence in BQL ............................ 36

3 Implementing BQL ........................................... 37
   3.1 Interface between BQL and Models ..................... 37
List of Figures

1-1 Evidence for an impending shortage of data scientists: Many analysts, including McKinsey, predict that the demand for data scientists will continue to rise much more rapidly than the supply of data scientists. Therefore, there is likely to be a huge demand for tools like BayesDB that enable non-experts to complete some data science tasks [39]..... 18

1-2 Comparison of traditional data analysis workflows to the BayesDB workflow .. 19

4-1 Latent Structure from [23], [32]: Naive Bayes (top), CRP Mixture (mid), CrossCat (bottom). Naive Bayes treats all columns as independent, and all rows are from the same component model. CRP Mixture still treats all columns as independent, but rows come from different underlying component models (categories). CrossCat groups columns into views (groups of dependent columns), each of which has its own different underlying categorization. 48

4-2 Venture code for Naive Bayes, DP Mixture, and CrossCat. A simple version of the Naive Bayes, DP Mixture, and CrossCat models for multinomial data in Venture. Note that Naive Bayes has no views or categories, DP Mixture has categories, and CrossCat has both views and categories. Venture is a general purpose probabilistic programming language: for details, see [22]. 50

5-1 BayesDB System Architecture: BayesDB has four main components: the frontend (which includes parser and client), the core BayesDB engine, an inference engine, and a prediction engine. The red arrow indicates a low-latency, low-volume connection, and the yellow arrows indicate a high-volume connection. 56

5-2 JavaScript Client: An initial prototype of what a web interface for BayesDB could look like. The JavaScript client simply needs to implement the same functionality as the Python client currently does, but there is a possibility to convert the user's mouse clicks on the UI into BQL queries that are sent to the backend, thereby bypassing the need for the user to type BQL at all. 57
6-1 An evaluation of BayesDB's ability to learn nonlinear relationships: The top row of blue plots show four original datasets, which each have 500 points and have 0 correlation. However, by visual inspection, it is easy to see that there is a different nonlinear relationship that exists in each of these datasets. Traditional statistics typically uses correlation to quantify relationships between variables, but it has a critical flaw: it is only able to measure linear relationships. The bottom row of plots shows the relationship that BayesDB learns from each original dataset. In each case, the original data was loaded, 32 models were initialized and analyzed for 500 iterations, and we called SIMULATE 500 times to generate 500 new data points from the joint probability distribution that BayesDB learned.

6-2 Testing BayesDB's ability to learn the full joint distribution: These plots compare CrossCat with other simpler bayesian models (CRP mixture model and Naive Bayes, with hyperparameter inference for each model) on a synthetic 300x20 dataset generated by CrossCat with 4 views and 8 categories per view using 20 models with 500 iterations of MCMC for each. The top plot was generated by evaluating the probability of a held-out row using the BQL PROBABILITY function: it shows that the mean probability of a held-out row is lower for more complex models, as expected. CrossCat's hypothesis space is a superset of CRP mixture's hypothesis space, which is a superset of Naive Bayes' hypothesis space. The bottom plot shows that the mean squared error (MSE) of the predictive probability over each value in the held-out row is lowest for CrossCat.

6-3 A comparison of missing value imputation accuracy between BayesDB and standard and sophisticated baselines: In this experiment, we took a 250 row, 100 column data table, removed a variable proportion of the data (x-axis): either 10%, 25%, 50%, 75%, or 90%, then loaded the data into BayesDB and generated 32 models with 300 iterations of analysis each, and used INFER to fill in all missing values with 500 samples from the posterior predictive for each missing value. CrossCat performs about as well as CRP Mixture at filling in missing data, but both are much better than Naive Bayes.
6-4 An empirical assessment of BayesDB’s calibration on synthetic datasets: These experiments were run on a synthetic data table with 500 rows, 4 columns, 2 views, and 8 categories per view, where each category’s data was generated from a multinomial distributed with 10 values. Afterwards, 20% of the data was randomly removed. The data was then loaded into BayesDB and analyzed with 32 models for 500 iterations. These three plots can be thought of as a multinomial P-P plot using normalized frequencies over missing values: each line represents the distribution of a column, where the y-axis is the inferred distribution and the x-axis is the actual distribution. These plots demonstrate that the three model types currently implemented in BayesDB (CrossCat, CRP mixture, and Naive Bayes) are all well calibrated.

6-5 Experimental evidence that BayesDB’s dependency detection is robust to the multiple comparisons problem: This experiment was run on a synthetic dataset with 100 rows and 32 or 64 columns, with two 2-column views. Therefore, the ground truth is that there are two pairs of columns that are dependent on each other, and all other pairs of columns are independent. This plot shows the dependence probability, as estimated by BayesDB using CrossCat, for all pairs of columns as a function of iterations. Each pair of columns is shown as a blue line, except for the two pairs of columns that are dependent according to ground truth. Before any iterations of inference, all pairs of columns start out with a fairly low yet uncertain dependence probability, accurately capturing BayesDB’s uncertainty. After a number of iterations, the dependence probabilities have converged to their true values, showing that BayesDB is able to discover individual relationships between variables in a robust way.

6-6 Empirical validation of CrossCat’s MCMC sampler using the Geweke test: The Geweke test is an inference diagnostic tool that compares the distributions of forward and backward sampled hyperparameters: in a correct sampler, the KL Divergence between those two distributions should converge to 0. The left plot shows a passing Geweke test, where all KL Divergences approach 0, and the right plot shows a failing Geweke test, where the KL Divergences do not all converge to 0 (this plot was used to diagnose a real bug in the sampler).
6-7 Estimates stabilize for DEPENDENCE PROBABILITY and INFER with more iterations: These plots show results on a real dataset: the Dartmouth Atlas of Healthcare, with 20% of the data randomly removed. The error bars shown in the plots are standard errors calculated over 5 runs, where each run has 20 models and 100 iterations of MCMC. In the top plot, the two lines represent the dependence probability between two pairs of columns: as the number of iterations increases, the estimates stabilize, and the probabilities move from 50% towards their true value. The bottom plot shows the proportion of the missing data that we are able to fill in with a confidence level of 10%: initially, only a small portion of the data is filled in, and the number of cells BayesDB is able to fill in at a given level of confidence increases with the number of iterations.

6-8 Timing Plots: Inference time (time per step, the y-axis in these plots) increases linearly in both the number of rows and number of columns.
List of Tables

1.1 BQL queries corresponding to specific statistical tasks: For any given statistical task, there are often a huge number of models, techniques, and algorithms that could be appropriate to use. This is the issue that scikit-learn tries to solve with their cheat sheet (see Figure 1-2c). Instead of a cheat sheet or complex workflow, BQL queries specify the task that should be solved, and the implementation of how that task is performed is abstracted as a black box to novice users (but is tunable/diagnosable for experts).

3.1 Which models implement which parts of the BQL Prediction Interface: This table shows the parts of the BQL interface different model classes are capable of implementing, including CrossCat, DPM (Dirichlet Process Mixture Model), NB (Naive Bayes), PGM (probabilistic graphical models), PGM+SL (probabilistic graphical models with structure learning), and KDE (kernel density estimation).

3.2 Implementation of BQL primitives: Each BQL query primitive is implemented using only one function in the BQL prediction interface. This table can be used in conjunction with Table 3.1 to determine which BQL queries are possible with each type of backend. In addition, the runtimes for BQL Primitives are given both in terms of number of calls to the predictive primitive (the functions in the prediction interface) and the overall runtime with the predictive primitive implemented by CrossCat (where `predictive_sample` runs in $O(1)$ for imputation and $O(q_c)$ for prediction, `predictive_probability` runs in $O(c)$, `column_dependence` runs in $O(1)$, and `row_similarity` runs in $O(q_c)$, as shown in Section 4.2.2). Let $s$ indicate the number of predictive samples parameter. Let $q_c$ indicate the number of queried columns and let $q_r$ indicate the number of queried rows (e.g. specified by a parameter in `SIMULATE`, or by the number of rows that matches the where clause in `INFER`). $c$ indicates the maximum number of categories over all $\Theta$ and $v$ denotes the maximum number of views over all $\Theta$. Let $d$ be the total number of columns, and let $r$ be the total number of rows.
Chapter 1

Introduction

1.1 Overview

BayesDB, a Bayesian database table, lets users query the probable implications of their tabular data as easily as an SQL database lets them query the data itself. Using the built-in Bayesian Query Language (BQL), users without statistics expertise can solve basic data science problems, such as detecting predictive relationships between variables, inferring missing values, simulating probable observations, and identifying statistically similar database entries.

BayesDB is suitable for analyzing complex, heterogeneous data tables with up to tens of thousands of rows and hundreds of variables. No preprocessing or parameter adjustment is required, though experts can override BayesDB’s default assumptions when appropriate.

BayesDB assumes that each row of the data table is a sample from some fixed population or generative process, and estimates the joint distribution over the columns. BQL’s estimates are provided by CrossCat, a nonparametric Bayesian model for analyzing high-dimensional data tables, but in theory any model, including graphical model structure learning, kernel density estimation, or naive Bayes, could implement BQL. This generality rests on the model independence provided by BQL, analogous to the physical data independence provided by the relational model.

This thesis provides a detailed description about the problem BayesDB solves - the difficulty of practical statistical data analysis - and how BayesDB was designed, architected, and implemented. An additional contribution of this thesis is the release of BayesDB as open source software under the Apache v2 license (download here: http://probcomp.csail.mit.edu/bayesdb/).
Demand In the United States for people with deep expertise in data analysis could be greater than its projected supply in 2018.

Deep analytical talent, thousands of FTEs

<table>
<thead>
<tr>
<th>Supply</th>
<th>Employment</th>
<th>Forecast of graduates</th>
<th>Net of additions through immigration and reemployment of previously unemployed</th>
<th>Projected 2018 supply</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006</td>
<td>156</td>
<td>+161</td>
<td>-32</td>
<td>285</td>
</tr>
</tbody>
</table>

2018 demand to realize full potential of big data

50-60% gap relative to supply given current trends, equal to 140,000-150,000 unfilled positions

"Deep analytical talent are people who have advanced training with statistics or machine learning. FTE = full-time equivalent.

Source: Dun & Bradstreet; company interviews; US Bureau of Labor Statistics; US Census Bureau; McKinsey Global Institute analysis

Figure 1-1: Evidence for an impending shortage of data scientists: Many analysts, including McKinsey, predict that the demand for data scientists will continue to rise much more rapidly than the supply of data scientists. Therefore, there is likely to be a huge demand for tools like BayesDB that enable non-experts to complete some data science tasks [39].

1.2 Motivation

The main goal of BayesDB is to address the immense complexity involved in standard statistical analysis. There are a wide variety of projects that aim to make machine learning accessible to non-expert users, but even these systems often require a reasonable level of skill to use. For example, examine the scikit-learn algorithm cheat sheet in Figure 1-2c and contrast it with the BayesDB workflow in Figure 1-2b. In order to meet the rapidly growing demand for data scientists, BayesDB is designed to make it easy for users with little statistics knowledge to answer basic data science questions.

Scikit-learn is a popular Python machine learning library that provides implementations of a huge number of common machine learning techniques. Even though scikit-learn alleviates the need for practitioners to implement custom machine learning algorithms in many cases, it does not eliminate the need for expertise.

The workflow for BayesDB is always the same, and there are no complex model selection or parameter tuning steps. However, BayesDB is not magic: the models do need to be learned from the data table after it is imported, and the INITIALIZE MODELS and ANALYZE steps can be time consuming because they are estimating the entire generative probabilistic process underlying the data table (although they do run in roughly $O(\text{size of data table})$, the constant factors are often slow enough to justify a lunch break).

BayesDB was developed with both novice and expert users in mind. Users with little or no formal statistics training certainly benefit the most from a tool like BayesDB, and we hope that it will increase the reliability of the empirical reasoning performed by non-experts who lack the expertise to efficiently build statistically sound models or critique the simplistic models that they do have access to. BayesDB will also be
BayesDB aims to drastically reduce the effort involved in the data understanding, modeling, and evaluation phases of this process.

BayesDB Workflow: Unlike the CRISP-DM workflow (a), BayesDB's workflow is designed to have no loops: data preparation and modeling phases (steps 1-3) only need to be done once. Step 4 replaces the data understanding and evaluation phases, and possibly even the deployment phase for certain environments. With this workflow, the need for anything like the scikit-learn cheat sheet (c) is completely alleviated.

(c) scikit-learn algorithm cheat sheet: This diagram is the cheat sheet for scikit-learn users who are trying to figure out what algorithm is appropriate for their problem. Even once the user navigates this confusing flowchart comprised of heuristics about which techniques work better in which settings, the user is still left to perform parameter tuning and cross validation on her own.

Figure 1-2: Comparison of traditional data analysis workflows to the BayesDB workflow
useful to users with statistics expertise who either have a problem that doesn't fit the standard supervised templates (e.g. the data is too sparse) or who want to analyze a tabular dataset but don't have the time or inclination for custom modeling. Figure 1-1 illustrates a standard data mining workflow which many practitioners follow today: even for experts, BayesDB could speed up each iteration through this workflow by eliminating the need to export data for analysis and implement custom models and evaluations.

Of course, when decisions can have costly consequences, there is no substitute for a critical assessment of the underlying rationale. In these situations, it is still necessary to get a real statistician to carefully approach the problem. Although BayesDB relies on unusually flexible assumptions, reports its uncertainty and is designed to err in the direction of under-confidence, its assumptions can still be misleading in some circumstances. Even in such high stakes settings, though, BayesDB is a powerful tool for rapid, effective exploratory data analysis.

1.3 Example data analysis

To demonstrate how BayesDB is used with real data, we will walk through an analysis of the Dartmouth Atlas of Healthcare dataset. It contains 307 hospitals with 67 attributes per hospital including variables measuring quality of care metrics, capacity metrics, cost of care, etc. [41]. The dataset is interesting in the field of healthcare economics because it was one of the first datasets that revealed how the quality of care at hospitals was unrelated to other factors including hospital budget and capacity. In a few BQL queries, we will replicate the main findings from this famous study.

First, the data must be loaded from a csv file:

```
bql> CREATE BTABLE dha FROM dha.csv;
Created btable dha. Inferred schema:
+-----------+--------------------------------------------------------------+---------------------------
|          | name             | n_death_ill          | ttl_mdcr_spnd | mdcr_spnd_inp | mdcr_spnd_outp |
+-----------+--------------------------------------------------------------+---------------------------
| multinomial | continuous | continuous | continuous | continuous | continuous |
```

When the data is loaded, BayesDB makes an informed guess about what the appropriate datatype is for each column. In this output, the first row shows column names, and the second row shows datatypes. For example, the `name` column is modeled as multinomial, which makes sense because names can't be represented as numbers, and `n_death_ill` is modeled as continuous because its values are all floating point numbers. If there are any mistakes, fix them with `UPDATE SCHEMA`.

At this point, the data is loaded, so all normal non-predictive queries will work, e.g. `SELECT`. However, we want to use predictive BQL queries, so we must initialize and analyze some models. The exact number of
Table 1.1: **BQL queries corresponding to specific statistical tasks**: For any given statistical task, there are often a huge number of models, techniques, and algorithms that could be appropriate to use. This is the issue that scikit-learn tries to solve with their cheat sheet (see Figure 1-2c). Instead of a cheat sheet or complex workflow, BQL queries specify the task that should be solved, and the implementation of how that task is performed is abstracted as a black box to novice users (but is tunable/diagnosable for experts).

<table>
<thead>
<tr>
<th>Statistical Task</th>
<th>Sample Standard Approaches</th>
<th>BQL Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-class Classification</td>
<td>One vs. All Classification [13] Ensemble methods, e.g. Random Forest [37] [30] Error Correcting Codes</td>
<td>INFER (categorical target)</td>
</tr>
<tr>
<td>Imputation</td>
<td>Mean Imputation Random Imputation Regression Imputation [9]</td>
<td>INFER</td>
</tr>
<tr>
<td>Answering hypotheticals</td>
<td>Custom approaches</td>
<td>SIMULATE</td>
</tr>
<tr>
<td>Predictive checks</td>
<td>Cross-Validation [17] Bootstrapping [17]</td>
<td>SIMULATE</td>
</tr>
<tr>
<td>Generating proxy datasets</td>
<td>Custom approaches [6]</td>
<td>SIMULATE</td>
</tr>
<tr>
<td>Query by example</td>
<td>Nearest Neighbor [3]</td>
<td>ORDER BY SIMILARITY TO example DESCENDING</td>
</tr>
<tr>
<td>Testing Dependence</td>
<td>Regression t-test or F-test [7] Correlation</td>
<td>DEPENDENCE PROBABILITY</td>
</tr>
<tr>
<td>Quantifying strength of linear predictive relationships</td>
<td>Correlation [28]</td>
<td>MUTUAL INFORMATION</td>
</tr>
</tbody>
</table>
models and iterations we use do not matter much, and if we leave them blank, BayesDB will use reasonable defaults.

```
bql> INITIALIZE 32 MODELS FOR dha;
Initialized 32 models.
```

```
bql> ANALYZE dha FOR 250 ITERATIONS;
Analyze complete.
```

Now that the models have been analyzed, we are ready to try some queries. The primary focus of this analysis is to understand what factors affect the quality of healthcare, which is stored in this table as `qual_score`. One of the most powerful exploratory queries in BQL is `ESTIMATE PAIRWISE DEPENDENCE PROBABILITY`, which computes the probability of each column being dependent on each other column.

```
bql> ESTIMATE PAIRWISE DEPENDENCE PROBABILITY FROM dha;
```
In this plot, every column is listed in the same order on the bottom axis as on the top axis, and the color of each cell indicates the dependence probability of each pair of columns. For example, the entire diagonal has dependence probability 1 because each column is always dependent on itself. The columns are ordered according to a hierarchical clustering so that groups of dependent columns are clustered together. The lower right portion of the matrix includes all four variables related to the quality of healthcare. Interestingly, almost all of the variables in the dataset have a dependence probability of 0 with healthcare quality. In one BQL query, we have learned that capacity and cost measures have absolutely nothing to do with healthcare quality.

To examine part of this dependence matrix more closely, we can look at the actual data among the 4 variables that are most dependent on qual_score:

```bql
PLOT SELECT (ESTIMATE COLUMNS FROM dha ORDER BY DEPENDENCE PROBABILITY WITH qual_score DESC LIMIT 4) FROM dha;
```

It is easy to see that indeed, the four columns that are very dependent with qual_score appear to have quite strong relationships with it.
Now that we have the sense that healthcare quality is probably not dependent on many other variables, if any, we can examine our dataset for hospitals that have unusual quality scores. First, as a diagnostic tool for predictive probability, we can plot how quality score relates to the predictive probability of quality score:

```
bql> PLOT SELECT qual_score, PREDICTIVE PROBABILITY OF qual_score FROM dha;
```

The results look reasonable: predictive probability of quality score is uncorrelated with quality score. In this plot, we can see an outlier with a particularly low predictive probability and a particularly low quality score. We can zoom in on it with the following query, which finds the hospitals that have the most unexpected qual_score given all other information about that hospital:

```
bql> SELECT name, qual_score, PREDICTIVE PROBABILITY OF qual_score FROM dha ORDER BY PREDICTIVE PROBABILITY OF qual_score LIMIT 5;
```

<table>
<thead>
<tr>
<th>name</th>
<th>qual_score</th>
<th>PREDICTIVE PROBABILITY OF qual_score</th>
</tr>
</thead>
<tbody>
<tr>
<td>McAllen TX</td>
<td>69.5</td>
<td>0.00031</td>
</tr>
<tr>
<td>Elyria OH</td>
<td>95.6</td>
<td>0.039</td>
</tr>
<tr>
<td>Minot ND</td>
<td>95.6</td>
<td>0.041</td>
</tr>
<tr>
<td>Lubbock TX</td>
<td>78.1</td>
<td>0.069</td>
</tr>
<tr>
<td>Fresno CA</td>
<td>82.0</td>
<td>0.087</td>
</tr>
</tbody>
</table>

This query reveals that McAllen, TX's qual_score was extremely unusual given McAllen's scores on other metrics: in fact, McAllen, TX had very high scores on cost and capacity metrics, yet still managed to
deliver a quite low quality of care. It is noteworthy that the predictive probability of McAllen's qual_score is two orders of magnitude lower than any other hospital. In fact, McAllen was so unusual that it was featured in a famous New Yorker article: “Patients in higher-spending regions received sixty per cent more care than elsewhere...yet they did no better than other patients” [8].

The true power of BayesDB is that with only a few queries, it is capable of delivering the same insights that previously were only possible to discover with the help of trained statisticians.

1.4 How does BayesDB work?

BQL is designed to encapsulate exploratory analysis and predictive modeling workflows in a way that is essentially independent of the underlying probabilistic modeling and inference approach. It turns analytic queries into approximate Bayesian inferences about the joint distribution induced by the underlying data generating process, and decouples that from the problem of building a model of that process. This is an inferential analogue of the physical data independence afforded by traditional DBMSes.

Although BayesDB is model-independent in theory, the current implementation's inferences are primarily based on CrossCat, which is a flexible, generic meta-model for probabilistic models of data tables that relaxes some of the unrealistic assumptions of typical Bayesian learning techniques. Approximate Bayesian inference in CrossCat tackles some of the model selection and parameter estimation issues that experts normally address by custom exploration, feature selection, model building and validation. Also, it produces estimates of the joint distribution of the data that is easy to conditionally query (unlike a Bayesian network) and that also has some useful latent variables. This makes several BQL commands natural to implement, and supports a "general purpose" BayesDB implementation. Just as traditional SQL-based DBMSs are specialized for different data shapes and workloads, usually for reasons of performance, we suspect BQL-based DBMSs could be specialized for reasons of both performance and predictive accuracy.

1.5 Outline

Chapter 2 gives examples of which BQL queries can be used to solve which statistical tasks, the formal language specification for BQL, and describes model independence in BQL. Chapter 3 describes how each BQL query is implemented, and gives the central interface between BQL and the underlying probabilistic models that provide BQL's model independence. Chapter 4 describes the models that are used to implement BayesDB, and shows in detail how CrossCat implements the BQL inference defined in Chapter 3. Chapter 5 focuses on the system architecture of BayesDB, including the client, engine, prediction/inference servers, how schemas are represented internally, and the query processing pipeline. Chapter 6 includes quality tests and experimental results demonstrating that BayesDB behaves reasonably in a wide variety of settings. Chapter
7 contains case studies of BayesDB being used to analyze real datasets. Chapter 8 describes related work in probabilistic databases and automated machine learning systems. Chapter 9 concludes and outlines areas for future research.
Chapter 2

Bayesian Query Language (BQL)

BQL is a SQL-like query language that adds support for running inference and executing predictive queries based on a bayesian model of the data. This chapter describes the formal language semantics of BQL, in addition to a primer on how to use BQL to solve standard statistical tasks, and includes a section that describes the analogy between database indexes in SQL and models in BQL.

2.1 Completing statistical tasks with BQL

The entire design goal of BQL is to make it easy to answer basic statistical questions. The language is designed to be as intuitive as possible to use (e.g. DEPENDENCE PROBABILITY is the function to use to determine whether two variables are dependent). The following brief subsections describe how to use BQL to solve a given statistical task, as initially outlined in Table 1.1.

2.1.1 Detect predictive relationships between variables and quantify their strength

BayesDB exposes the existence of predictive relationships between variables with the DEPENDENCE PROBABILITY function. In order to figure out which variables are most dependent on column x, use: ESTIMATE COLUMNS FROM table ORDER BY DEPENDENCE PROBABILITY WITH x DESC, which will return all columns (variables) from the table, ordered by how likely it is that there exists a dependence between x and that variable. In order to view how each variable interacts with all others, use ESTIMATE PAIRWISE DEPENDENCE PROBABILITY FROM table, which displays a matrix of the dependence probability of each variable with every other variable.

In order to quantify these relationships, we can use the same queries as above, but with MUTUAL INFORMATION instead of DEPENDENCE PROBABILITY, which only evaluates the presence or absence (not the strength) of a relationship. Mutual information is a measure in the same spirit as correlation, but is able to express more complex nonlinear dependencies, while correlation only measures linear relationships.
2.1.2 Identify contextually similar database entries

SIMILARITY returns the similarity between two rows. It can be used to find similar rows to a specific row (query by example), e.g. to find people similar to the person named John Smith in your database: SELECT * FROM table ORDER BY SIMILARITY TO ROW WHERE name='John Smith'.

In addition to computing overall similarity, similarity can also be computed for subsets of columns. For example, this query finds which people are closest to John Smith according to their demographic info - their age, race, zip code, and income: SELECT * FROM table ORDER BY SIMILARITY TO ROW WHERE name='John Smith' WITH RESPECT TO age, race, zip_code, income.

2.1.3 Infer missing values with appropriate error bars

When some values are missing from the database, there are many scenarios in which it would be helpful to fill them in. Sometimes data is missing simply because it could not be collected, e.g. a survey respondent refusing to answer a particular question, and sometimes it is missing because the value cannot be known for sure until the future. INFER can be useful both for prediction, and to facilitate other downstream analyses that require all values to be filled in (to satisfy those methods, it is common practice to fill in missing values with sloppy heuristics like mean or median imputation). In either case, INFER is able to fill in missing values as long as the model is sufficiently confident. For example, a column that indicates whether a patient has disease X could be filled in with INFER disease.x FROM table WITH CONFIDENCE 0.95, which will fill in a value for whether the patient probably has disease x, as long as the probability that the value is correct is at least 95% according to BayesDB's model.

2.1.4 Generate hypothetical rows conditioned on arbitrary predicates

To gain an understanding of what other types of data points are probable given the observed data, an analyst may be interested in simulating new data points. For example, suppose there is a dataset of web browsing sessions, but only a relatively small number of them are from people who visited a download page on their cell phones. BayesDB is capable of generating new probable data points, conditioned on that predicate: SIMULATE * FROM table GIVEN browser='cell phone' AND visited.download='True'.

Another very important use case for SIMULATE is to alleviate privacy constraints: if the contents of the raw data table are private, in some settings it may still be possible to release aggregates of the data table as long as they preserve privacy. SIMULATE can generate new data points which preserve nearly all important statistical properties from the original dataset, but are completely synthetic in order to satisfy privacy concerns.
2.2 Formal Language Description

This section provides a description of all BQL language constructs, in roughly the order that they might be used to analyze real data with BayesDB.

2.2.1 Loading Data

- **CREATE BTABLE `<btable>` FROM `<filename.csv>`**

  Creates a btable by importing data from the specified CSV file. The file must be in CSV format, and the first line must be a header indicating the names of each column.

- **UPDATE SCHEMA FOR `<btable>` SET `<col1>=<type1>[,<col2>=<type2>...]**

  Types are categorical (multinomial), numerical (continuous), ignore, and key. Key types are ignored for inference, but can be used later to uniquely identify rows instead of using ID. Note that datatypes cannot be updated once the model has been analyzed.

- **EXECUTE FILE `<filename.bql>`**

  It is possible to write BQL commands in a file and run them all at once by using EXECUTE FILE. This is especially useful with UPDATE SCHEMA for tables with many columns, where it can be a good idea to write a long, cumbersome UPDATE SCHEMA query in a separate file to preserve it.

2.2.2 Initializing Models and Running Analysis

It is necessary to run INITIALIZE MODELS and ANALYZE in order for BayesDB to evaluate any query involving any predictions. The models that this command creates can be thought of as possible explanations for the underlying structure of the data in a Bayesian probability model. In order to generate reasonable predictions, initializing roughly 32 models works well in most cases (although, if computation is a bottleneck, reasonably good results can be achieved with about 10 models), and 250 iterations is a sensible amount of iterations to run ANALYZE to achieve good convergence. The more models created, the higher quality the predictions will be, so if it is computationally feasible to initialize and analyze with more models or iterations, there is no downside (other than the extra computation). While there are reasonable default settings for the average user, advanced users have diagnostic tools (see SHOW DIAGNOSTICS) that they may use to assess convergence to determine how many models and iterations are needed in their specific case.

- **INITIALIZE `<num_models>` MODELS FOR `<btable>`**

  Initializes `<num_models>` models.
• **ANALYZE** <btable> [MODELS <model_index>-<model_index>] FOR (<num_iterations> ITERATIONS | <seconds> SECONDS)

Analyze the specified models for the specified number of iterations (by default, analyze all models).

### 2.2.3 Btable Administration

A Btable is a Bayesian data table: it comprises the raw data, the data types for each column (the schema), and the models. There are a few convenience commands available to view the internal state of BayesDB:

- **LIST BTABLES**
  View the list of all btable names in BayesDB.

- **SHOW SCHEMA FOR** <btable>
  View each column name, and each columns datatype.

- **SHOW MODELS FOR** <btable>
  Display all models, their ids, and how many iterations of **ANALYZE** have been performed on each one.

- **SHOW DIAGNOSTICS FOR** <btable>
  Advanced feature: show diagnostic information for a given Btable's models.

- **DROP BTABLE** <btable>
  Deletes an entire btable (including all its associated models).

- **DROP MODEL[S]** [<model_index>-<model_index>] FROM <btable>
  Deletes the specified set of models from a Btable.

### 2.2.4 Saving and Loading Btables and Models

Save and load models allow models to be exported from one instance of BayesDB (save), and then imported back into any instance of BayesDB (load), so that the potentially time-consuming **ANALYZE** step does not need to be re-run.

- **LOAD MODELS** <filename.pkl.gz> INTO <btable>

- **SAVE MODELS** FROM <btable> TO <filename.pkl.gz>
2.2.5 Querying

BayesDB has five fundamental query statements: SELECT, INFER, SIMULATE, ESTIMATE COLUMNS, and ESTIMATE PAIRWISE. They bear a strong resemblance to SQL queries.

- **SELECT <columns|functions> FROM <btable> [WHERE <whereclause>]**
  
  [ORDER BY <columns|functions>] [LIMIT <limit>]

  SELECT is just like SQL’s SELECT, except in addition to selecting, filtering (with where), and ordering raw column values, it is also possible to use predictive functions in any of those clauses.

- **INFER <columns|functions> FROM <btable> [WHERE <whereclause>]**
  
  [WITH CONFIDENCE <confidence>] [WITH <numsamples> SAMPLES]
  
  [ORDER BY <columns|functions>] [LIMIT <limit>]

  INFER is just like SELECT, except that it also attempts to fill in missing values. The user must specify the desired confidence level to use (a number between 0 and 1, where 0 means fill in every missing value no matter what, and 1 means only fill in a missing value if there is no uncertainty). Optionally, the user may specify the number of samples to use when filling in missing values: the default value is good in general, but expert users who want higher accuracy can increase the number of samples used.

- **SIMULATE <columns> FROM <btable> [WHERE <whereclause>] TIMES <times>**

  SIMULATE generates new rows from the underlying probability model a specified number of times.

- **ESTIMATE COLUMNS FROM <btable> [WHERE <whereclause>] [ORDER BY <functions>]**
  
  [LIMIT <limit>] [AS <column_list>]

  ESTIMATE COLUMNS is like a SELECT statement, but selects columns instead of rows.

- **ESTIMATE PAIRWISE <function> FROM <btable> [FOR <columns>] [SAVE TO <file>]**
  
  [SAVE CONNECTED COMPONENTS WITH THRESHOLD <threshold> AS <column_list>]

  ESTIMATE PAIRWISE can evaluate any function that takes two columns as input, i.e. DEPENDENCE PROBABILITY, CORRELATION, or MUTUAL INFORMATION, and generate a matrix showing the value of that function applied to each pair of columns. See the Predictive Functions section for more information.

  The optional clause SAVE CONNECTED COMPONENTS WITH THRESHOLD <threshold> AS <column_list> may be added in order to compute groups of columns, where the value of the pairwise function is at least <threshold> between at least one pair of columns in the group. Then, those groups of columns can be saved as column lists with names column_list_<id>, where id is an integer starting with 0:

- **ESTIMATE PAIRWISE ROW SIMILARITY FROM <btable> [FOR <rows>] [SAVE TO <file>]**
  
  [SAVE CONNECTED COMPONENTS WITH THRESHOLD <threshold> [AS] <btable>]
Compute pairwise functions of rows with ESTIMATE PAIRWISE ROW.

In the above query specifications, <columns|functions> indicates a list of comma-separated column names or function specifications, e.g. name, age, TYPICALITY, date. <whereclause> indicates an AND-separated list of <column> <operator> <value>, where operator must be one of (=, <, <=, >, >), e.g. WHERE name = 'Bob' AND age <= 18 AND TYPICALITY > 0.5.

2.2.6 Query Modifiers

SUMMARIZE or PLOT may be prepended to any query that returns table-formatted output (almost every query) in order to return a summary of the data table instead of the raw data itself, e.g. SUMMARIZE SELECT * FROM table. This is extremely useful as a tool to quickly understand a huge result set: it quickly becomes impossible to see trends in data by eye without the assistance of SUMMARIZE or PLOT.

- SUMMARIZE displays summary statistics of each of the output columns: for numerical data, it displays information like the mean, standard deviation, min, and max, and for categorical data it displays the most common values and their probabilities.

- PLOT displays plots of the marginal distributions of every single output column, as well as the joint distributions of every pair of output columns. PLOT displays a heat map for pairs of numerical columns, the exact joint distribution for pairs of categorical columns, and a series of box plots for mixed numerical/categorical data. Many tools, like R and pandas, have functionality similar to PLOT when all the data is the same type, but PLOT is specially designed and implemented from the ground up to behave well with mixed datatypes.

2.2.7 Column Lists

Instead of manually typing in a comma-separated list of columns for queries, it is possible to instead use a column list in any query that asks for a list of columns. Column lists are created with ESTIMATE COLUMNS, which allows columns to be filtered with a where clause, ordered, limited in size, and saved by giving them a name with the AS clause:

- ESTIMATE COLUMNS FROM <btable> [WHERE <whereclause>] [ORDER BY <functions>] [LIMIT <limit>] [AS <column_list>]

Note that the functions that may be used in the WHERE and ORDER BY clauses for ESTIMATE COLUMNS are different from normal queries: please refer to Section 2.2.8 for details of which functions can be used in which queries, and see below for an example of using functions of columns:
ESTIMATE COLUMNS FROM table WHERE TYPICALITY > 0.6
ORDER BY DEPENDENCE PROBABILITY WITH name;

- SHOW COLUMN LISTS FOR <btable>
  Print out the names of the stored column lists in the btable.

- SHOW COLUMNS <column_list> FROM <btable>
  View the columns in a given column list.

2.2.8 Predictive Functions

Functions of rows

Functions that take a row as input may be used in many types of queries, including SELECT, INFER, ORDER BY (except in ESTIMATE COLUMNS), and WHERE (except in ESTIMATE COLUMNS). Functions in this category include:

- **SIMILARITY TO <row> [WITH RESPECT TO <column>]**
  Similarity measures the similarity between two rows as the probability that the two rows were generated by the same underlying probabilistic process. By default, similarity considers all columns when deciding how similar two rows are, but it is possible to optionally specify a set of one or more columns to compute similarity with respect to only those columns.

- **TYPICALITY**
  The typicality of a row measures how structurally similar to other rows this row is: typicality is the opposite of anomalousness, and anomalousness is defined as the probability that the row is in a cluster (i.e. of a latent variable model) on its own. Rows with low typicality can be thought of as outliers. If the underlying structure is a single high-entropy cluster, typicality will be high yet predictive probability will be low. If the underlying structure is many low-entropy clusters, typicality will be low yet predictive probability will be high.

- **PROBABILITY OF <column>=<value>**
  The probability of a cell taking on a particular value is the probability that the underlying probability model assigns to this particular outcome. For continuous columns, though, it instead returns the value of the probability density function evaluated at value.

- **PREDICTIVE PROBABILITY OF <column>**
The predictive probability of a value is similar to the \text{PROBABILITY OF} \text{<column>=<value>} query, but it measures the probability that each cell takes on its observed value, as opposed to a specific value that the user specifies. It answers the question: if it were to observe this column in this row again, what is the probability of the observed value? See \text{TYPICALITY} above for a comparison between it and \text{PREDICTIVE PROBABILITY}.

Here are is an example of these functions used in a \text{SELECT} query:

- \text{SELECT SIMILARITY TO ROW 0 WITH RESPECT TO name, TYPICALITY FROM btable WHERE PROBABILITY OF name='Bob' > 0.8 ORDER BY PREDICTIVE PROBABILITY OF name;}

\textbf{Functions of two columns}

Functions of two columns may be used in \text{ESTIMATE PAIRWISE} (omit the 'OF' clause) and \text{SELECT} (include the 'OF' clause; they only return one row). If used in a \text{SELECT} query along with other functions that return many rows, then the same value will be repeated for each of those rows. The three functions of this type are described below:

- \text{DEPENDENCE PROBABILITY [OF <column1> WITH <column2>]}  
The dependence probability between two columns is a measure of how likely it is that the two columns are dependent (opposite of independent). Note that this does not measure the strength of the relationship between the two columns; it merely measures the probability that there is any relationship at all.

- \text{MUTUAL INFORMATION [OF <column1> WITH <column2>]}  
  Mutual information between two columns measures how much information knowing the value of one column provides about the value in the other column: it is a way to quantify the strength of a predictive relationship between variables that works even for nonlinear relationships. If mutual information is 0, then knowing the first column provides no information about the other column (they are independent). Mutual information is always nonnegative, and is measured in bits.

- \text{CORRELATION [OF <column1> WITH <column2>]}  
  This is the standard Pearson correlation coefficient between the two columns. All rows with missing values in either or both of the two columns will be removed before calculating the correlation coefficient.

Here are examples that demonstrate how to use these functions in different query types:

- \text{ESTIMATE PAIRWISE DEPENDENCE PROBABILITY OF name WITH age;}
- \text{SELECT MUTUAL INFORMATION OF name WITH age FROM table...}
Functions of one column (for SELECT)

Functions in this category take one column as input, and can only be used in SELECT (but they only return one row). If used in a query along with other functions that return many rows, then the same value will be repeated for each of those rows.

There is only one function like this:

- **TYPICALITY OF** `<column>`

  The typicality of a column measures how structurally similar to other columns this column is: typicality is the opposite of anomalousness, and anomalousness is defined as the probability that the column is in a view (i.e. of a latent variable model) on its own. Columns with low typicality can be thought of as independent of all or most other columns.

And here is an example of how to use it:

- **SELECT TYPICALITY OF age FROM...**

Functions of one column, for ESTIMATE COLUMNS

For each of the functions of one or two columns above (that were usable in SELECT, and sometimes ESTIMATE PAIRWISE), there is a version of the function that is usable in ESTIMATE COLUMNS’s WHERE and ORDER BY clauses:

- **TYPICALITY**

  This is the same function as TYPICALITY OF `<column>` above, but the column argument is implicit.

- **CORRELATION WITH** `<column>`

  This is the same function as CORRELATION OF `<column1>` WITH `<column2>` above, but one of the column arguments is implicit.

- **DEPENDENCE PROBABILITY WITH** `<column>`

  This is the same function as DEPENDENCE PROBABILITY OF `<column1>` WITH `<column2>` above, but one of the column arguments is implicit.

- **MUTUAL INFORMATION WITH** `<column>`

  This is the same function as MUTUAL INFORMATION OF `<column1>` WITH `<column2>` above, but one of the column arguments is implicit.

Here is an example of this type of function used in an ESTIMATE COLUMNS query:

- **ESTIMATE COLUMNS FROM table WHERE TYPICALITY > 0.6 AND CORRELATION WITH name > 0.5 ORDER BY DEPENDENCE PROBABILITY WITH name;**
2.3 Model Independence in BQL

A model in BQL is in many ways analogous to an index in SQL. Just as SQL hides the representation of data, BQL hides the representation of the model: the user only interacts with models through a fairly high level interface (see Section 3.1 for details).

Before SQL, in the era of hierarchical and network data models, database programmers were forced to go through the cumbersome process of writing specific algorithms to access data for every individual query. Currently, data scientists need to implement and train specific models for every individual statistical query of their data. Consider running custom regressions to fill in missing values: for every new target variable or set of features, a new regression must be trained. Analogously to SQL, the goal of BQL is to eliminate the need for data scientists to implement and train models: a BayesDB model can fill in missing values with INFER without any custom implementation or re-training.

Just as expert users of SQL can specify an index to be created to optimize a particular task, expert users of BQL can specify a specific model to optimize a particular statistics task. For example, if the user is particularly interested in predicting one specific column with extremely high accuracy, then she could train a custom regression model to accomplish that one specific task.

Just as indexes perform better than a naive for loop, except when for loops already provide optimal performance, CrossCat performs better than Naive Bayes (NB) or Dirichlet Process Mixture (DPM, or DP mixture), except when those are able to perfectly characterize the latent generative probabilistic process. Naive Bayes (with hyperparameter inference) embeds into DP mixtures, which embeds into CrossCat, on which we perform fully Bayesian inference. Therefore with enough data, if either DPM or NB were better than CrossCat, exact samples from the CrossCat posterior would be (essentially) no worse in accuracy and performance than exact samples from DPM or NB, because they would yield (essentially) the same distribution on latent structures. In practice this occurs with very small datasets and the wiggle room for “essentially” (the amount of data that cannot be predicted from since it is being used to select the NB or DPM subset from the larger CrossCat hypothesis set) is indistinguishable.
Chapter 3

Implementing BQL

The idea behind BQL is that we can formalize many data analysis tasks using Bayesian queries against a posterior distribution over a hypothesis space that itself consists of models of the full joint distribution and underlying structure of the variables in the table. This mathematical notion lets us define a coherent interface to the implications of data that retains the clarity, simplicity and flexibility of SQL. It is our analogue of the relational algebra.

In our current implementation of BayesDB, a single probabilistic model, called CrossCat, is estimated from the data, where each sample produced by ANALYZE is independently drawn from a distribution between the CrossCat prior and the true posterior given the data in the table. The longer ANALYZE is run, the closer to the true posterior BayesDB becomes. The more samples from the posterior there are, the more accurately posterior uncertainty can be resolved.

3.1 Interface between BQL and Models

Throughout this chapter, we use \( \Theta \) to indicate a 'model', or 'Bayesian Index', of the data. A 'model' of the data is assumed to be a sample from the posterior distribution: \( p(\Theta|X) \), where \( X \) is the entire dataset (all observed data points), and \( \Theta \) includes all parameters and hyperparameters of the model: \( \Theta = \{\theta, \alpha\} \). The posterior predictive is the distribution that new data points are sampled from, conditioned on the observed data:

\[
p(x'|X) = \int_\Theta p(x'|\Theta)p(\Theta|X)d\Theta
\]

It is assumed that we do not have the analytic form of the posterior \( p(\Theta|X) \), but that we do have the ability to approximately sample it. Since the approximate samples \( \hat{\Theta} \) we obtain from the posterior are a
result of MCMC inference, they are a function of time $T$ and converge to the true posterior as inference time increases: $p(\hat{\Theta}(T)|X) \rightarrow p(\Theta|X)$ as $T \rightarrow \infty$. Therefore, the above integral is impossible to evaluate. Instead, we have a set of $m$ models $\{\Theta\} = \{\Theta_1, \ldots, \Theta_m\}$, which we use to compute a Monte Carlo approximation of the posterior predictive:

$$p(x'|X) \approx \frac{1}{m} \sum_{\Theta} p(x'|\Theta)p(\Theta|X) \approx \frac{1}{m} \sum_{\Theta} p(x'|\hat{\Theta})p(\hat{\Theta}|X)$$

Additionally, we are often interested in the posterior predictive, conditioned on an additional set of predicates $Y$. $Y$ is necessary to implement the GIVEN clause in SIMULATE, and many other queries. The posterior predictive conditioned on $Y$ is as follows:

$$p(x'|X,Y) = \int_{\Theta} p(x'|\Theta,Y)p(\Theta|X)d\Theta$$

$$p(x'|X,Y) \approx \frac{1}{m} \sum_{\Theta} p(x'|\Theta,Y)p(\Theta|X) \approx \frac{1}{m} \sum_{\hat{\Theta}} p(x'|\hat{\Theta},Y)p(\hat{\Theta}|X)$$

Therefore, it is necessary that any model used to implement BayesDB supports approximate sampling from $p(x'|\Theta,Y)$.

### 3.1.1 Indexing Interface

The purpose of the indexing interface is to provide a function, or functions, that allow BayesDB to generate an index of new models $\hat{\Theta}$ that are approximate samples from $p(\Theta|X)$. In practice, the indexing interface consists of two functions: `initialize`, which samples $\Theta$ from the prior, and `analyze`, which transforms $\hat{\Theta}$ to iteratively be closer and closer to being a true sample from $p(\Theta|X)$. In practice, certain classes of models do not have a need for `analyze`. For example, the implementation of a kernel density estimate (KDE) model would have an `initialize` implementation that generates $p(\Theta|X)$, and `analyze` would simply do nothing. In fact, the only invariant that this interface must satisfy is that `initialize` must generate a model $\Theta$, and `analyze` must iteratively move $\Theta$ closer to being a true sample from $p(\Theta|X)$.

- **initialize($M_c, M_r, T$) → $\hat{\Theta}_i$**
  Initializes a new model. The subscript $i$ is used to indicate that this is an individual new model, not a set of models. `initialize` should return a model sampled from the prior: $\Theta_i \sim p(\Theta)$.

- **analyze($M_c, T, \hat{\Theta}_i, iters$) → $\hat{\Theta}'_i$**
  Analyze takes a model as an argument, performs a certain number of MCMC transitions on the model, and then returns the new, modified model. Each iteration of `analyze` brings $\hat{\Theta}_i$ closer to being a true
sample from the posterior: As the number of iterations of analyze, $T$, grows large, then analyze should return a model that is a good approximate sample from the posterior: $p(\hat{\Theta}(T)|X) \rightarrow p(\Theta|X)$ as $T \rightarrow \infty$.

### 3.1.2 Prediction Interface

The prediction interface is at the core of BQL's implementation: these are the functions that BQL calls in order to answer predictive queries of the data. The prediction interface answers queries based on the assumption that the underlying model, $\Theta$, is true. Section 3.2 details how these functions based on a single $\Theta$ can be aggregated across all models, $\{\Theta\}$, in order to provide more accurate estimates than any that could be obtained from a single model.

This is a unique approach to inference. Often in probabilistic databases, or statistics in general, a single model $\Theta$ is learned, and all inferences are based solely on that model. This is what makes BQL the Bayesian Query Language; it incorporates integration over models given the observed data in the table. It also supports queries about the latent structure, thereby supporting a broader class of analytical tasks than would be handled by a Predictive Query Language. In the interface description below, arguments in angled brackets ($<>$) are optional.

- **predictive_sample**($M_c, \Theta_t, \{Y\}, columns, row$) → $x'$

  Generates a sample, $x'$, from the posterior predictive distribution for the given column: $x \sim p(x'|X,Y)$.

- **predictive_probability**($M_c, \Theta_t, \{Y\}, columns, <row>, value$) → $p$

  Computes the predictive probability that a sample drawn from the specified column, conditioned on $Y$, will be equal to the specified value: $p(x' = value|X,Y)$.

- **row_similarity**($M_c, \Theta_t, r_1, r_2, columns$) → $s$

  For each column in the list of arguments, row similarity is 1 if the column was generated by the same underlying component model according to $\Theta$ for each of the two given rows, and 0 if not. Then, the 0's and 1's for each column in the list of arguments are averaged to compute an overall similarity score.

- **column_dependence**($M_c, \Theta_t, c_1, c_2$) → $d$

  Returns either 0 or 1, indicating whether the two specified columns are dependent in $\Theta$. In order to get more granular estimates of column dependence, the values of this function are averaged over multiple $\Theta$s.
Table 3.1: **Which models implement which parts of the BQL Prediction Interface:** This table shows the parts of the BQL interface different model classes are capable of implementing, including CrossCat, DPM (Dirichlet Process Mixture Model), NB (Naive Bayes), PGM (probabilistic graphical models), PGM+SL (probabilistic graphical models with structure learning), and KDE (kernel density estimation).

<table>
<thead>
<tr>
<th></th>
<th>CrossCat</th>
<th>DPMM</th>
<th>NB</th>
<th>PGM</th>
<th>PGM+SL</th>
<th>KDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>predictive_sample</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>predictive_probability</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>row_similarity</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>column_dependence</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>Heterogeneous Datatypes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>Linear-time approximate inference</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>Linear-time prediction</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 3.2: **Implementation of BQL primitives:** Each BQL query primitive is implemented using only one function in the BQL prediction interface. This table can be used in conjunction with Table 3.1 to determine which BQL queries are possible with each type of backend. In addition, the runtimes for BQL Primitives are given both in terms of number of calls to the predictive primitive (the functions in the prediction interface) and the overall runtime with the predictive primitive implemented by CrossCat (where predictive_sample runs in $O(1)$ for imputation and $O(q_c)$ for prediction, predictive_probability runs in $O(c)$, column_dependence runs in $O(1)$, and row_similarity runs in $O(q_c)$, as shown in Section 4.2.2). Let $s$ indicate the number of predictive samples parameter. Let $q_c$ indicate the number of queried columns and let $q_r$ indicate the number of queried rows (e.g. specified by a parameter in SIMULATE, or by the number of rows that matches the where clause in INFER). $c$ indicates the maximum number of categories over all $\Theta$ and $v$ denotes the maximum number of views over all $\Theta$. Let $d$ be the total number of columns, and let $r$ be the total number of rows.

<table>
<thead>
<tr>
<th>BQL Primitive</th>
<th>Predictive Primitive</th>
<th>Calls to Pred. Primitive</th>
<th>CrossCat Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMULATE</td>
<td>predictive_sample</td>
<td>$O(1)$</td>
<td>$O(q_r,q_c)$</td>
</tr>
<tr>
<td>INFER</td>
<td>predictive_sample</td>
<td>$O(s)$</td>
<td>$O(sq_r,q_c)$</td>
</tr>
<tr>
<td>MUTUAL INFORMATION</td>
<td>predictive_sample</td>
<td>$O(s)$</td>
<td>$O(sq_r,q_c)$</td>
</tr>
<tr>
<td>PROBABILITY OF col=val</td>
<td>predictive_probability</td>
<td>$O(q_c)$</td>
<td>$O(q_c)$</td>
</tr>
<tr>
<td>PREDICTIVE PROBABILITY</td>
<td>predictive.probability</td>
<td>$O(q_r)$</td>
<td>$O(q_r)$</td>
</tr>
<tr>
<td>DEPENDENCE PROBABILITY</td>
<td>column_dependence</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>COLUMN TYPICALITY</td>
<td>column_dependence</td>
<td>$O(d)$</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>SIMILARITY</td>
<td>row_similarity</td>
<td>$O(1)$</td>
<td>$O(q_r)$</td>
</tr>
<tr>
<td>ROW TYPICALITY</td>
<td>row_similarity</td>
<td>$O(r)$</td>
<td>$O(rd)$</td>
</tr>
</tbody>
</table>
3.2 Implementation of Query Primitives

The following section describes both the math and the algorithms that are used to implement every query in BQL. These algorithms frequently make use of the API between BQL and \( \Theta \), as defined in the previous section.

3.2.1 SIMULATE

SIMULATE generates new hypothetical data points according to the underlying models of the data that BayesDB has learned. The distribution to sample from is \( p(x'|X, Y) \), the posterior predictive conditioned on the GIVEN clause.

To sample an individual datapoint, we first sample a particular model \( \Theta_k \), then sample the datapoint from the posterior predictive given that one particular model, \( p(x'|\Theta_k,Y) \).

\[
\begin{align*}
  & k \sim \text{uniform-discrete}(1,m) \\
  & x' \sim p(x'|\Theta_k,Y)
\end{align*}
\]

Algorithm 1: simulate

\[
\begin{align*}
  \text{input} & : X, Y, \{\Theta\}, M_c, \text{column} \\
  \text{output} & : x' \\
  1 & \Theta_i \leftarrow \text{uniform}(\{\Theta\}) \\
  2 & x' \leftarrow \text{predictive.sample}(M_c, \Theta_i, Y, \text{column}, \text{nrows + 1})
\end{align*}
\]

3.2.2 INFER

INFER fills in a missing value with a point estimate, \( e_i \), over its predictive distribution given the observed data \( X \), marginalized out models \( \Theta \). Note that this is not the predictive distribution \( p(x'|X) \), but rather \( p(x_r,c_0,...,x_r,c_k|X) \), where \( r \) is the current row id and \( c_0...c_k \) are the columns mentioned in the targets and the condition (WHERE clause), together. The point estimate \( e_i \) is often the mode, median, or mean, depending on the datatype.

However, INFER doesn’t always fill in missing values: it only fills in missing values when its confidence, \( c \), is above the specified threshold. For data with discrete outcomes, confidence is defined as \( p(x_r,c_0,...,x_r,c_k|X)'s probability mass on the inferred values:

\[
c = p(x_r,c_0 = e_0,...,x_r,c_k = e_k|X)
\]
And when data is continuous, it is necessary to set a tolerance parameter $\epsilon$, since the probability mass

on any specific value $e$ will always be 0. $\epsilon$ should be set so that its scale is proportional to the scale of the
data.

**Algorithm 2:** infer (discrete)

<table>
<thead>
<tr>
<th>input</th>
<th>: $X, {\Theta}, M_c, {x_r, c_0, ..., x_r, c_k}, n, c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>: $x'$</td>
</tr>
</tbody>
</table>

1. $x \leftarrow []$
2. $Y \leftarrow T[\text{row}]$ // set constraints to be the other observed values in row
3. for $n$ iterations do
   4. $\Theta_i \leftarrow \text{uniform} (\{\Theta\})$
   5. $x.\text{append} (\text{predictive.sample} (M_c, \Theta_i, Y, \{x_r, c_0, ..., x_r, c_k\}, \text{row}))$
   6. end
7. $m \leftarrow x.\text{mode}()$
8. if $x.\text{count}(m)/n \geq c$ then
   9. return $m$
10. end
11. return None

### 3.2.3 WHERE clauses WITH CONFIDENCE

A WHERE clause that specifies a specific confidence level using WITH CONFIDENCE is very similar to INFERENCE. In fact, if the predicate in the WHERE clause tests for equality, then the implementation is exactly the same as INFER: the value is filled in with $\epsilon$ if the confidence $c$ is at least equal to the specified amount.

The interesting difference between WHERE clauses and INFER is that WHERE doesn’t have to fill in a particular value, it just needs to decide whether the predicate is satisfied, so inequalities are possible in WHERE clauses. However, the implementation is different when the WHERE clause tests inequalities. Consider the

where clause $x' > e$: we are interested in computing $c = p(x' > e|\Theta_k)$, and then averaging over models by sampling from each of them.

$$c = p(x' > e|\Theta_k) = \int_{\epsilon}^{\infty} p(x' = s|\Theta_k) ds$$

This integral is approximated with $n$ samples, where the predicate $P(x')$ is the predicate specified by the

WHERE clause (in this case $P(x') = (x' > e)$):

$$c \approx \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}(P(x_i)), \quad x_i \sim p(x'|\Theta_k)$$

Note that we do not simply compute the mean or mode of $x'$ and determine whether that value satisfies the predicate. We must individually evaluate the predictive for every sample from $p(x'|\Theta_k)$ to correct for
cases where the mean or mode satisfies the predicate, but the bulk of the probability mass does not. As
an example, consider the predicate: WHERE x > 100 AND x < 200. If we sample from \( p(x'|\Theta_k) \) 100 times,
and receive 50 samples where \( x' = 0 \) and 50 samples where \( x' = 300 \), then the predicate is never satisfied in
reality, but \( E[x'] = 150 \) does satisfy the predicate.

---

**Algorithm 3:** where

```
input : X, \{\Theta\}, M_c, row, column, p, c: where p is the predicate, and c is the confidence
output : boolean: whether predicates are satisfied
1  count ← 0
2  for n iterations do
3     \( \Theta_t \leftarrow \text{uniform}(\{\Theta\}) \)
4     \( x' \leftarrow \text{predictive_sample}(M_c, \Theta_t, Y, \text{column}, \text{row}) \)
5     if \( p(x') \) then
6         count ← count + 1
7     end
8  end
9  return (count ≥ c)
```

---

### 3.2.4 ESTIMATE COLUMNS

ESTIMATE COLUMNS requires no special magic: it executes just like a SELECT query, except it allows columns
to be selected instead of rows, and uses functions of columns in WHERE and ORDER BY clauses.

### 3.2.5 ESTIMATE PAIRWISE

ESTIMATE PAIRWISE simply calls the specified pairwise function (of either two rows, or two columns) on
each pair (or rows or columns) in the dataset. E.g., ESTIMATE PAIRWISE DEPENDENCE PROBABILITY just
involves computing DEPENDENCE PROBABILITY between each pair of columns, then reordering the columns
in the output according to an agglomerative clustering’s dendogram in order to arrange clusters of similar
columns together. The runtime required is \( O(c) \), where \( c \) is the number of columns in a pairwise column
operation or the number of rows in a pairwise row operation.

### SAVE CONNECTED COMPONENTS

Connected components is a method to extract hard clusterings from a pairwise matrix. Without loss of
generality, consider a pairwise function of columns, instead of rows (although they would both be implemented
analogously). Consider the graph \( G \) where each vertex \( c \) is a column, and every pair of columns \( c_i, c_j \) has
an edge \( e_{i,j} \) with weight \( w_{i,j} \), where \( w_{i,j} \) is the value of the pairwise function that was computed between
columns, e.g. DEPENDENCE PROBABILITY. SAVE CONNECTED COMPONENTS takes a threshold \( t \) as an argument
and partitions the graph into groups of columns where all edges between groups have weight less than \( t \), and
each column $c_i$ in a group is connected to at least one other column $c_j$ in the same group with an edge that has weight $w_{i,j} \geq t$.

This procedure can be implemented $O(c + e)$ time, where $d + e$ is the number of edges between columns, with a depth first search, simply adding columns greedily to any existing components $e$ is upper bounded by $c^2$ because a fully connected graph has $c^2$ edges, so the upper bound on runtime is $O(c^2)$, which is equivalent to the runtime of ESTIMATE PAIRWISE. Similarly, the pairwise row version of save connected components has runtime $O(r + e)$, where $e$ is the number of edges between rows, which is upper bounded by $O(r^2)$.

3.3 Functions

3.3.1 SIMILARITY

SIMILARITY is directly implemented by the row.similarity function in the BQL prediction interface. Similarity compares two rows, and returns the probability that the two rows were generated by the same underlying probabilistic process. In a mixture model with latent variables, as CrossCat is, it is equivalent to asking what the probability is that these two data points were generated from the same setting of the latent variable (equivalently, the probability that they are samples from the same component model).

3.3.2 DEPENDENCE PROBABILITY

DEPENDENCE PROBABILITY is directly implemented by the column.dependence function in the BQL prediction interface. Two columns are dependent if either column affects the other in the underlying generative process in $\Theta$.

3.3.3 MUTUAL INFORMATION

MUTUAL INFORMATION is a function that quantifies dependence between two variables, which represents the information gained about one variable by conditioning on the other. It is defined as follows:

$$MI_{\Theta}(i, j) = E_{X_i, X_j|s_e} \left[ \log \left( \frac{p(x_i, x_j|\Theta)}{p(x_i|\Theta)p(x_j|\Theta)} \right) \right]$$

Where $\Theta$ represents a single sample from CrossCat’s posterior, and $x_i$ and $x_j$ are the two variables whose mutual information is being computed. The Monte Carlo estimate of the mutual information which is used by BayesDB is defined as follows:

$$MI_{\Theta}(i, j) \approx \frac{1}{s} \sum_{n=1}^{s} \log \left( \frac{p(x_i', x_j'|\Theta)}{p(x_i'|\Theta)p(x_j'|\Theta)} \right)$$
Where we use \( s \) predictive samples to approximate the true posterior predictive, and where \( x'_i \) and \( x'_j \) are predictive samples from an unobserved row (e.g. they are generated using the \texttt{SIMULATE} algorithm).

### 3.3.4 TYPICALITY (row)

The **Typicality** of a row is equal to the probability that a row is not the only one of its kind: it is essentially 1 minus the probability that it is an outlier. For a categorization of rows \( r \) into categories \( c \), with \( n \) rows, typicality is defined as follows:

\[
t(r_i) = 1 - \frac{1}{n-1} \sum_{j \neq i}^{n} p(c(r_i) \neq c(r_j))
\]

### 3.3.5 TYPICALITY (column)

Column **Typicality** is analogous to row typicality: for a categorization of columns \( c \) into views \( v \), with \( d \) columns, **Typicality** is defined as follows:

\[
t(c_i) = 1 - \frac{1}{d-1} \sum_{j \neq i}^{d} p(v(c_i) \neq v(c_j))
\]

### 3.3.6 PROBABILITY OF \( \text{col}=<\text{val}> \)

**Probability** is a very simple function to evaluate: it is simply the value of the posterior predictive at a certain value, averaged over models. For discrete data, **Probability** of \( x=v \) is evaluated as:

\[
p(x' = v | X) = \mathbb{E}_\Theta p(x' = v | \Theta)
\]

And for continuous data, the value of the probability density function is returned instead (so in those cases, it is in fact possible for this function to return values greater than 1).

### 3.3.7 PREDICTIVE PROBABILITY

**Predictive Probability** is computed in the same way as **Probability** of \( \text{col}=<\text{val}> \), except \( <\text{val}> \) is different for each row, and is equal to the observed value of \( \text{col} \) in that row. **Predictive Probability** is undefined when there is no observed value.
Chapter 4

Indexer and Models

Up until now, this thesis has treated BayesDB models, \( \{ \Theta \} \), as a black box that must implement the API defined in 3.1. A large part of the beauty of BayesDB is that it does define that API, so that the rest of the system design can be isolated from details of how the individual models work. However, in order to have a solid understanding of the entire system, this chapter goes into detail on how a model might implement the BayesDB API.

4.1 Model Descriptions

This section describes the models that are currently used to implement BayesDB, and briefly outlines other types of models that might work well.

4.1.1 CrossCat, CRP Mixture, and Naive Bayes

Right now, BayesDB uses CrossCat to estimate the joint distribution over the columns, and performs posterior averaging over these estimates to answer all BQL queries. CrossCat is more flexible than Naive Bayes: it determines which variables are probably independent from the data, as opposed to assuming they all are independent a priori, and estimates a flexible nonparametric Bayesian density model (which we will refer to as “component model”) for each group of dependent variables.

More formally, CrossCat is a Dirichlet process mixture model (over the columns) of Dirichlet process mixture models (DPMs) over the rows. It can be understood as a nonparametric Bayesian take on the problem that Bayesian network structure learning tries to solve. The outer DPM groups columns into “views” - groups of columns that are structurally dependent. Each inner DPM (there is one per view) groups the rows in each view into “categories”. To see how a sample dataset might be partitioned into views and categories in a single CrossCat sample, see Figure 4-1.
Figure 4-1: Latent Structure from [23], [32]: Naive Bayes (top), CRP Mixture (mid), CrossCat (bottom).
Naive Bayes treats all columns as independent, and all rows are from the same component model. CRP Mixture still treats all columns as independent, but rows come from different underlying component models (categories). CrossCat groups columns into views (groups of dependent columns), each of which has its own different underlying categorization.
CrossCat was originally developed by the computational cognitive science community as a probabilistic model meant to describe cross-categorizations in data tables. Its extremely good properties both as a tool to understand the latent structure of the data and as a prediction engine make it an ideal backend for BayesDB [23], [32], [33].

The three models that are currently implemented in BayesDB (Naive Bayes, CRP Mixture (also known as DP Mixture), and CrossCat) are all generative probabilistic models, which means that each model specifies the underlying probabilistic process from which data points are assumed to be generated. Naive Bayes’s hypothesis space is a subset of CRP Mixture’s, which is a subset of CrossCat’s. Figure 4-1 illustrates how the hypothesis spaces of these three models are nested. CRP Mixture is a special case of CrossCat, which occurs when all columns are in the same view, and Naive Bayes is a special case of CRP Mixture, which occurs when all rows are in the same category.

In all three models, we use datatype-appropriate component models with the appropriate conjugate prior and hyperparameter inference, so as a result, Naive Bayes as implemented in BayesDB is more robust than a typical implementation of Naive Bayes, which often has no priors on component models and no hyperparameter inference.

4.1.2 Component Models and Data Types

The production version of BayesDB currently has built-in types for categorical and generic numerical data, modeled by the Dirichlet-Multinomial and Normal-Inverse-Gamma component models, respectively. A wide range of datatypes are possible, both by remaining within the exponential family - for e.g. count data, ordinal data, positive real data, and more - and by going beyond it, at the cost of slower inference. Unstructured data fields, such as text, can be converted to structured formats with preprocessing steps such as keyword extraction, and structured compound fields, such as time series in individual cells, must be modeled with more complex models than CrossCat, such as the Time CrossCat variation which models each individual cell as a time series. One great advantage of models like Naive Bayes, DP Mixture, and CrossCat is that they are extremely flexible in terms of which types of component models are possible. Any probability distribution can be implemented as a datatype for CrossCat, and in a version of CrossCat implemented by a probabilistic programming language like Venture (see Figure 4-2 for one such implementation), it is particularly easy to prototype such component models.

4.1.3 Incorporating Prior Knowledge

Some say that the real strength of Bayesian modeling is about incorporating prior knowledge. If that is true, why doesn’t BayesDB allow users to incorporate prior knowledge? In fact, some Bayesian priors are about precisely expressing ignorance. This is what CrossCat is doing inside BayesDB: encoding a very broad prior
Naive Bayes

[ASSUME get-component.hyperparameter (mem (lambda (col) (gamma 1.0 1.0)))]
[ASSUME get-component.model (mem (lambda (col)
    (make_sym.dir_mult (get-component.hyperparameter col) num.values)))]
[ASSUME get-cell (mem (lambda (row col)
    ((get-component.model col))))]
For each row, column observed pair (r,c):
[OBSERVE get.cell r c]

DP Mixture

[ASSUME get-component.hyperparameter (mem (lambda (col) (gamma 1.0 1.0)))]
[ASSUME get-component.model (mem (lambda (col category)
    (make_sym.dir_mult (get-component.hyperparameter col) num.values)))]
[ASSUME get-categorization.crp
    (make_crp (gamma 1.0 1.0))]
[ASSUME get.category (mem (lambda (view row)
    ((get-categorization.crp view))))]
[ASSUME get-cell (mem (lambda (row col)
    ((get-component.model col (get-category row)))))]
For each row, column observed pair (r,c):
[OBSERVE get.cell r c]

CrossCat

[ASSUME get-component.hyperparameter (mem (lambda (col) (gamma 1.0 1.0)))]
[ASSUME get-component.model (mem (lambda (col category)
    (make_sym.dir.mult (get-component.hyperparameter col) num.values)))]
[ASSUME view.crp.hyperparameter (gamma 1.0 1.0)]
[ASSUME view.crp (make_crp view.crp.hyperparameter)]
[ASSUME get.view (mem (lambda (col) (view.crp)))]
[ASSUME get.categorization.crp.hyperparameter
    (mem (lambda (view) (gamma 1.0 1.0)))]
[ASSUME get.categorization.crp (mem (lambda (view)
    (make_crp (get.categorization.crp.hyperparameter view))))]
[ASSUME get.category (mem (lambda (row)
    ((get-categorization.crp view))))]
[ASSUME get-cell (mem (lambda (row col)
    ((get-component.model col (get-category (get.view col) row)))))]
For each row, column observed pair (r,c):
[OBSERVE get.cell r c]

Figure 4-2: Venture code for Naive Bayes, DP Mixture, and CrossCat. A simple version of the Naive Bayes, DP Mixture, and CrossCat models for multinomial data in Venture. Note that Naive Bayes has no views or categories, DP Mixture has categories, and CrossCat has both views and categories. Venture is a general purpose probabilistic programming language: for details, see [22].

over generative processes that admits possibilities ranging from near determinism to unpredictability.

That said, BQL does allow users to encode some of their prior knowledge via schema design. Currently this means deciding what the columns and rows in the table are, and what the datatypes are for each column. Picking data types for each column is equivalent to picking the probabilistic model used for that datatype. For example, consider a “time of day” variable, which is represented as an integer from 0 to 1439 (the number
of minutes in a day). Without incorporating prior knowledge, a Normal-Inverse-Gamma model (the default for numerical data) would be used. However, according to that model, 0 is very far from 1439, when in reality midnight is only one minute away from 11:59pm. This could be remedied by the user selecting a von Mises distribution (cyclical normal) [1] to be used to model the “time of day” column.

If it survives, BQL will likely evolve towards becoming a general-purpose probabilistic programming language, perhaps even one like Venture or Church. From this perspective, the current CrossCat backend corresponds to a simple kind of Bayesian probabilistic program synthesis, where the class of probabilistic programs it is learning is extremely simple.

4.1.4 Other Models

BQL can be implemented by a huge variety of models - the models don’t need to be natively Bayesian, as Naive Bayes, DP Mixture, and CrossCat are. In fact, it is perfectly fine to use a kernel density estimate (KDE), graphical models, or even a typical machine learning prediction algorithm like random forests or SVMs. Of course, while CrossCat and other similar models are able to implement every function of the BQL predictive interface, most other models can only implement certain parts of the interface. Table 3.1 shows which parts of the BQL interface some example models can implement. One reasonable approach could be to obtain a large number of different types of models, and use some specialized models to answer one specific type of query particularly well. For example, in the standard regression/classification setting (INFER with a single target column, large N, and where all other columns for all rows are fully observed) the accuracy could sometimes be improved by integrating discriminative modeling techniques. For example, a random forest could be trained to always answer queries that need to predict one particular column given a set of other particular columns, and CrossCat could be used for all other predictive queries. BayesDB backends based on deep learning and/or density forests might be interesting for very large datasets, especially ones with unstructured (text, images, video, etc.) columns.

4.2 CrossCat’s Implementation of BQL Interface

Recall from the previous chapter that for any model used to implement BayesDB, each model Θ (stored in CrossCat as X_L and X_D, described in Section 5.3.1) must implement the prediction interface described in Section 3.1.2.

This section describes in detail how a CrossCat model implements each of these functions: initialize, analyze, predictive.sample, predictive.probability, row.similarity, and column.similarity. Each of these functions can be executed in linear time by CrossCat. initialize and analyze are categorized as “offline inference”; they must be called after loading the data into BayesDB before doing anything else.
The rest of the functions are categorized as “online predictions”, because those four functions power all of BayesDB’s predictions.

4.2.1 Inference Functions

Initialize

initialize\( (M_c, M_r, T) \rightarrow \Theta_i \)

Initialize samples a model \( \Theta \) from the prior, \( p(\Theta) \). This is done by simply forward sampling from each distribution in the generative process: sample hyperparameters from their hyperpriors, sample a clustering of columns into views from a CRP, etc. Sampling from the prior is a much faster operation than analyze in the following section. For a full discussion of the CrossCat generative model, see [23].

Analyze

analyze\( (M_c, T, \Theta_i) \rightarrow \Theta_i \)

Analyze performs one MCMC transition of \( \Theta \). An MCMC transition involves starting with a given latent structure (as pictured in Figure 4-1), and applying a series of transition operators. There are three transition operators (MCMC kernels) that are cycling through. First, resample the CRP concentration hyperparameters \( \alpha \) and \( \{\alpha_v\} \) from \( \alpha \sim p(\alpha|z) \). Next, the component model hyperparameters, \( \{\lambda_d\} \) are each resampled from \( p(\lambda_d|d, y^z) \). Then, the category assignments of rows are resampled for each row in each view, according to the likelihood of that row under each category’s component model. Finally, each column’s view is resampled. In total, all of the above transitions run in \( O(rows \times columns \times (categories + views)) \). In practice, \( categories + views \) is often much smaller than \( rows \) or \( columns \), so inference runs roughly linearly in the number of cells in the data table. As inference in CrossCat is not a focus of this thesis, please see [23] for a full treatment.

4.2.2 Prediction Functions

Predictive Sample

CrossCat implements predictive sample by sampling a category for each view, then updating the component model for that category to satisfy the constraints in \( Y \) and sampling from the constrained component model (see Algorithm 4).

Predictive Probability

CrossCat implements the predictive probability of a queried cell by iterating over each category in the queried cell’s view and adding the log probability of \( x \) in each constrained component model (see Algorithm 5).
Algorithm 4: predictive_sample

```
input : \( M_c, X_L, X_D, Y, r, D \).
output : A list of sampled values \( x \).

// Separate conditions, \( Y \), by whether they apply to the queried row, \( r \)
1 \( Y^q \leftarrow Y_{s.t.} Y_{s.1} = r \)
2 \( Y^o \leftarrow Y_{s.t.} Y_{s.1}! = r \)
3 \( \text{views}_{\text{sampling}} \leftarrow \{ \text{get_view}(X_L, d) \}_{d \in D} \)
4 foreach \( \text{view}_i \in \text{views}_{\text{sampling}} \) do
5     if \( r \) denotes an observed row then
6         // The category is deterministically set
7             \( c_{\text{view}_i} \leftarrow \text{get_category}(X_D, \text{view}_i, r) \)
8     else
9         // sample a category, allowing the possibility of a new category
10            \( Pr(c|\text{view}, Y^q, Y^o, X_L) \propto Pr(Y^q|c, \text{view}, Y^o, X_L)Pr(c|\text{view}, X_L) \)
11             \( c_{\text{view}_i} \sim Pr(c|\text{view}_i, Y^q, Y^o, X_L) \)
12     end
13 foreach \( d \in D \) do
14     \( \text{view}_i \leftarrow \text{get_view}(X_L, d) \)
15     \( \text{cm} \leftarrow \text{get_component_model}(X_L, \text{view}_i, c_{\text{view}_i}, d) \)
16     // Determine conditions that modify this component model
17     \( Y^c \leftarrow Y_{s.t.} Y_{s.1} = d \) and get_category\( (X_D, \text{view}_i, Y_{s.2}) = c_{\text{view}_i} \)
18     // Update the component model
19     \( \text{cm}' \leftarrow \text{update_component_model}(\text{cm}, Y^c) \)
20     // Sample a value from the posterior predictive (PP) distribution
21     \( x_i \sim PP(\text{cm}') \)
22 end
```

Row Similarity

Row similarity with respect to a particular column is computed as the proportion of CrossCat latent states \( X_L \) whether the two rows are in the same category in the particular column's view. Row similarity without respect to a particular column averages that quantity over all columns (see Algorithm 6).

Column Dependence

Column dependence is implemented in CrossCat by directly reading off whether the two columns are in the same view (see Algorithm 7).
Algorithm 5: predictive_probability

input : $M_c, X_L, X_D, Y, Q$
output : A list of marginal probabilities $p$

foreach $i \in Q$ do
    Extract the queried row, dimension, and value:
    $r, d, x \leftarrow Q_i$
    Find all elements of $Y$ that apply to the same dimension:
    $$Y' \leftarrow Y \text{s.t. } Y_{i,2} = d$$
    $p_i \leftarrow 0$
    foreach categories in view (and a new category) do
        if isContinuous($d$) then
            $pX_c = cdf(x + \epsilon|X_L, Y') - cdf(x - \epsilon|X_L, Y')$
        end
        $pX_c = P(x|X_L, Y')$
        $pCat_c = P(z_d = c|Y')$
    end
    $pCat = \logsumexp(pCat)$
    $p_i = \sum_c (pX \times pCat)$
end

Algorithm 6: row_similarity

input : $M_c, X_L, X_D, r_1, r_2, columns$
output : $p$

$s \leftarrow 0$
foreach $c$ in columns do
    if $get.category(X_D, get.view(X_L, c), r_1) == get.category(X_D, get.view(X_L, c), r_2)$ then
        $s \leftarrow s + 1$
end
return $s / \text{len(columns)}$

Algorithm 7: column_dependence

input : $M_c, X_L, X_D, c_1, c_2$
output : $p$

return $get.view(X_L, c_1) == get.view(X_L, c_2)$
Chapter 5

System Design and Architecture

This chapter dives deeper into the system design issues that were involved in building BayesDB, with a focus on the system architecture. BayesDB is implemented as a standalone database, with no dependencies on any other databases: it includes its own parser for BQL, its own native Python, command line, and (prototype) HTML/JavaScript clients, query processor, persistence layer, and inference/prediction servers. BayesDB was originally developed as a layer on top of Postgres, but it quickly became apparent that BayesDB required such a drastic change to the system architecture that it needed to be built as an entirely new system. This chapter details about each specific component of the system, as shown in Figure 5-1.

5.1 Client

The client is the piece of software that the user interacts with. The BayesDB client is written entirely in Python, so it can be used as follows:

```
$ python
>>> from bayesdb.client import Client
>>> c = Client()
>>> c('INFER * FROM table WITH CONFIDENCE 0.95;')
```

All functionality is accessible through this Python client object, but users are used to interacting with databases through their own standalone command line tools, so BayesDB has one such tool of its own. When BayesDB is installed (like a normal Python package, with $ python setup.py install), it installs the bql command line tool:

```
$ bql
Welcome to BayesDB. You may enter BQL commands directly into this prompt. Type 'help' for help, and 'quit' to quit.
```
Figure 5-1: BayesDB System Architecture: BayesDB has four main components: the frontend (which includes parser and client), the core BayesDB engine, an inference engine, and a prediction engine. The red arrow indicates a low-latency, low-volume connection, and the yellow arrows indicate a high-volume connection.

```
bql> INFER * FROM table WITH CONFIDENCE 0.95;
```

Under the hood, the client calls the parser to parse each query, loads and saves data to/from disk, displays matplotlib graphics, and calls the BayesDB engine.

There is no reason the client needs to be implemented in Python. In fact, there is a prototype BayesDB web client, with a command line interface directly in the browser, and an easy interface for uploading data and viewing visualizations (see Figure 5-2).

5.2 Parser

The parser is a Python module whose core function is `parse(input_string)`, which returns the appropriate BayesDB engine function to call and with what arguments, and also a list of arguments to the client. The client uses those arguments so it can perform local operations immediately before or after the query is issued to the BayesDB engine, e.g. the client may have to load a file from disk and send it to a remote BayesDB engine.

Nested queries are parsed all at once and sent to the engine in a single batch. The engine executes the
5.3 Engine

The BayesDB Engine is where the bulk of the computation occurs. All query processing algorithms from Section 3 are executed by this engine.

5.3.1 Data and Metadata

This section describes the metadata stored by BayesDB about a data table: $T$, the entire raw data table, $M_c$, the column metadata, and $M_r$, the row metadata. Additionally, BayesDB stores a set of models $\{\Theta\}$, which are internally represented as $X_L$ and $X_D$ for CrossCat.

$M_c$ is a dict that contains three keys: \texttt{name.to.idx, idx.to.name}, and \texttt{column.metadata}. \texttt{name.to.idx} and \texttt{idx.to.name} convert from column names to indices and indices to column names, respectively.
column metadata is a list, where each entry is a dict with three keys: modeltype, value_to_code, and code_to_value. modeltype maps to a string containing one of five possible model types: ‘asymmetric_beta_bernoulli’, ‘normal_inverse_gamma’, ‘pitmanyor_atom’, ‘symmetric_dirichlet_discrete’, and ‘poisson_gamma’. value_to_code maps between values and how they are encoded in the data. For instance, for the ‘binary’ data type, this might map ‘True’ to 1 and ‘False’ to 0. For the ‘categorical’ data types, this would function similarly, but with more possible values. For both ‘categorical’ and ‘binary’ data types, values must start from zero and increment up by 1. code_to_value maps back from codes to values. For the ‘normal_inverse_gamma’ and ‘poisson_gamma’ model types, value_to_code and code_to_value are empty because there is no conversion to be done.

$M_r$ is a list, where entries are dicts with two keys: name_to_idx and idx_to_name. These map from row names to row indices and back.

For instance, consider the following data table:

<table>
<thead>
<tr>
<th>Height(in)</th>
<th>Gender</th>
<th>Nationality</th>
<th>IQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>66</td>
<td>Male</td>
<td>US</td>
</tr>
<tr>
<td>Steve</td>
<td>?</td>
<td>Male</td>
<td>Canadian</td>
</tr>
<tr>
<td>Jill</td>
<td>60</td>
<td>Female</td>
<td>US</td>
</tr>
</tbody>
</table>

$M_c$ and $M_r$ could be:

```
M_c: {
    "name_to_idx": {0: "Height", 1: "Gender", 2: "Nationality", 3: "IQ"},
    "idx_to_name": {"Height":0, "Gender":1, "Nationality":2, "IQ":3},
    "column_metadata": [ {
        "modeltype": "normal_inverse.gamma",
        "value_to_code": {},
        "code_to_value": {}}, {
        "modeltype": "asymmetric_beta_bernoulli",
        "value_to_code": {0:"Male", 1:"Female"},
        "code_to_value": {"Male":0, "Female":1}}, {
        "modeltype": "symmetric_dirichlet_discrete",
        "code_to_value": {"US":0, "Canadian":1},
        "value_to_code": {1:"Canadian", 0:"US"}}, {
        "modeltype": "normal_inverse_gamma",
        "value_to_code": {},
        "code_to_value": {}}
    ]
```

58
M_r: {
    "name_to_idx": {"Bob":0, "Steve":1, "Jill":2},
    "idx_to_name": {0:"Bob", 1:"Steve", 3:"Jill"},
}

The latent state, $X_L$, is a dict with three keys: column_partition, column_hypers, and view_state. column_partition is a dict with three keys: hypers, assignments, and counts. hypers is a dict containing a single key, alpha, which maps to a number representing the concentration parameter of the CRP. assignments is an array, where each element is an integer representing the view to which that column is assigned. counts is an array with length equal to the number of views, where each entry is the number of columns in that view.

column_hypers is an array where each element is a dict whose structure depends on the modeltype of that column. For instance, for a column with a 'asymmetric_beta_bernoulli' modeltype, the dict would have two keys: strength and balance. One additional key would apply to all model types. fixed would map to the strings 'true' or 'false' to indicate whether the parameters were fixed or not.

view_state is an array where each element is a dict with three keys: column_names, row_partition_model and column_component_suffstats. column_names is an array of strings where the indices are locally consistent (within the view).

row_partition_model is a dict with two keys: hypers and counts. hypers is a dict with a single key, alpha, which maps to a number representing the concentration parameter for the row CRP in that view. counts is an array, with length equal to the number of categories, where each element is the number of rows assigned to that category. column_component_suffstats is an array of arrays of dicts, where the structure of each dict depends on the model type. The outer array is over the columns while the inner array is over categories. Each dict contains the sufficient statistics that are appropriate for the model type of the column. For instance, for an 'asymmetric_beta_bernoulli' column, the dict would have three keys: 0_count, 1_count, and N. Continuous model types' suffstats must include a value N for the count of the number of non-empty values.

Assume the underlying crosscat state is:

views: [1 1 2 3]
categories: [[1 1 2] [1 2 1] [1 1 1]]
Then, \( X_L \) could be:

\[
X_L: \{
    "column_partition": {
        "hypers": {"alpha": 1.1},
        "assignments": [0, 0, 1, 2],
        "counts": [2, 1, 1],
    },
    "column_hypers": [
        {"fixed": false, "mu": 63.5, "kappa": 8.2, "alpha": 2.3, "beta": 3.4},
        {"fixed": false, "strength": 10, "balance": 1.2}
    ],
    "view_state": [
        {"row_partition_model": {"hypers": {"alpha": 3.2}, "counts": [2, 1]}}, ...
    ]
}
\]

The mapping between the latent state and the data, \( X_D \), is an array of arrays. The elements of the outer array correspond to views and the inner array corresponds to the rows. Entries are integers indicating, for a row in a view, which category it belongs to.
5.3.2 Query Processing Pipeline

At a high level, all predictive BQL queries - SELECT, INFERENCE, and SIMULATE - are processed through the same query processing pipeline. Other queries are either single-purpose administrative commands (e.g. UPDATE SCHEMA), or are other single-purpose commands (e.g. ESTIMATE PAIRWISE doesn’t have a complex query processing pipeline: it simply determines which columns to use from the FOR COLUMNS clause, then evaluates the specified function on each pair of columns).

Algorithm 8: Query Processing Pipeline

```plaintext
input : INFERENCE q FROM T WHERE w ORDER BY o LIMIT l: q, w, o, l, T.
output : T'.

// Evaluate WHERE clause (w) and filter T based on which rows match
foreach r in T do
  if not w(r) then
    T.remove(r)
  end
end

// Evaluate ORDER BY clause (o) and reorder T according to it
foreach r in T do
  r[key] ← o(r)
end
T.sort(key=lambda r: r[key])
// Truncate T according to limit
T ← T[:l]
// Evaluate query (q)
T' ← []
foreach r in T do
  T'.append(q(r))
end
return T'
```

Algorithm 8 gives an overview of the rough order of operations that take place in the main query processing pipeline. This algorithm is quite simplified by its ability to abstract the entire implementation of a WHERE clause as \( w() \) and the evaluation of all queried values and functions as \( q() \). For the details of how \( q() \) and \( w() \) are implemented, see Section 3.2. In order to avoid excess computation, for example, \( w() \) is applied first, then \( o() \) is only computed on rows that satisfied \( w() \). Then, \( l() \) is applied, so that \( q() \) is only evaluated for rows
that are actually returned to the user in the final output. Note again that there is a further simplification in the above algorithm: if the same function appears in some combination of $w()$, $o()$, and $q()$, it is computed the first time and then cached for future evaluations, to avoid evaluating predictive functions multiple times.

### 5.3.3 Reliable Distributed Indexing and Model Balancing

In order to avoid situations where remote analyze processes die and BayesDB never updates the corresponding model again, the BayesDB Engine now regularly polls the inference engine to look for orphaned processes. If one is found (presumably due to being out of memory), the analyze call is added into a queue in the BayesDB Engine. This congestion control strategy is similar to TCP's slow-start algorithm. BayesDB sends as many analyzes as it can to the inference engine until one dies; then it doesn’t resend any more analyzes until a previous one completes.

Each call to analyze runs only one iteration of MCMC because the computation time required is many orders of magnitude greater than the communication time, even over slow networks. This allows the BayesDB Engine to maintain a set of models $\{\Theta\}$, and update each individual model asynchronously in the background every time one of the analyze calls returns. From the user’s perspective, this asynchronous background model-updating is invisible. All the user sees is that over time, the same predictive queries start returning results with tighter error bars and higher accuracy, until convergence is reached.

### 5.4 Inference and Prediction Servers

The BayesDB Engine server interacts with the inference and prediction servers exclusively through the interface between BQL and models defined in Section 3.1. Inference servers execute initialize and analyze calls, and prediction servers execute predictive_sample, predictive_probability, row_similarity, and column_dependence. Currently, BayesDB uses CrossCat inference and prediction servers, but BayesDB would also work if these servers supported other types of models as well. Oftentimes, for smaller setups, the inference and prediction servers are running on the same machine.
Chapter 6

Experiments and Quality Tests

In this chapter, we show the results of many experiments that evaluate BayesDB’s performance and accuracy across a range of both real datasets and synthetic datasets (datasets that were generated from CrossCat, with some true but unknown parameters).

6.1 BayesDB’s query results are trustworthy in many cases

Bayesian inference, Bayesian nonparametrics, careful hyperparameter inference and Markov chain Monte Carlo are incredible tools, but they are not a panacea. However, BayesDB is reliable enough to trust as a first pass on typical data tables because CrossCat has some unusual properties that make it a reasonable general-purpose model:

- CrossCat contains various simpler, standard black box models as hypotheses in its hypothesis space - including Dirichlet process mixtures and a form of naive Bayes. This means that with enough inference, CrossCat should not perform essentially worse than these simpler methods, and is likely to perform better.

- CrossCat includes many hypotheses that correspond to versions of the claim “I can find no signal in this part of the dataset”, and it reports broad predictive distributions when it is uncertain. Although there might actually be signal that another method could have found, we think this is a good direction to err in. Some versions have been tested with the kinds of small N, high-D, extremely noisy datasets that confuse many standard methods and performed surprisingly well [23].

- There are theoretical results that suggest CrossCat is an asymptotically consistent estimator of the joint distribution [23]. This has actually been proved for Dirichlet process mixture models with various component models, and the increased flexibility of CrossCat should not interfere (as N → ∞, though
Figure 6-1: **An evaluation of BayesDB’s ability to learn nonlinear relationships:** The top row of blue plots show four original datasets, which each have 500 points and have 0 correlation. However, by visual inspection, it is easy to see that there is a different nonlinear relationship that exists in each of these datasets. Traditional statistics typically uses correlation to quantify relationships between variables, but it has a critical flaw: it is only able to measure linear relationships. The bottom row of plots shows the relationship that BayesDB learns from each original dataset. In each case, the original data was loaded, 32 models were initialized and analyzed for 500 iterations, and we called SIMULATE 500 times to generate 500 new data points from the joint probability distribution that BayesDB learned.

there is a subtlety if D → ∞). This roughly means that for a surprisingly broad class of distributions, if inference is run long enough, and there is enough data, CrossCat will infer a model that predicts equivalently with the ‘true’ model.

### 6.1.1 Limitations of CrossCat, the current BayesDB backend

Nearly deterministic relationships between variables are easy for CrossCat to detect, but awkwardly modeled: a large number of low-variance clusters are introduced, yielding a kind of adaptively-tuned nearest neighbor.

In principle, CrossCat can estimate any joint distribution given enough data, even in the presence of multiple conflicting statistical signals in different groups of columns. In practice, this might require unrealistic amounts of data and computation. There is future work to be done to better characterize the regimes where CrossCat is getting highly accurate (and properly uncertain) estimates, and regimes where another backend
would be more appropriate. Possibilities include exploring heterogeneous backends, model ensembles, and richer formulations of model selection for reducing these limitations.

6.2 Convergence Assessment

What are the consequences of trusting the output of BayesDB after running “too few iterations”? Markov Chain Monte Carlo techniques all have the fundamental problem of convergence assessment: how many iterations do we need to wait before we are sampling from the true posterior? BayesDB partially alleviates these concerns by initializing the chains from the prior, which causes initial estimates to express uncertainty (e.g. a dependence probability far from 0 or 1) and have large error bars.
Figure 6-2: Testing BayesDB’s ability to learn the full joint distribution: These plots compare CrossCat with other simpler bayesian models (CRP mixture model and Naive Bayes, with hyperparameter inference for each model) on a synthetic 300x20 dataset generated by CrossCat with 4 views and 8 categories per view using 20 models with 500 iterations of MCMC for each. The top plot was generated by evaluating the probability of a held-out row using the BQL PROBABILITY function: it shows that the mean probability of a held-out row is lower for more complex models, as expected. CrossCat’s hypothesis space is a superset of CRP mixture’s hypothesis space, which is a superset of Naive Bayes’ hypothesis space. The bottom plot shows that the mean squared error (MSE) of the predictive probability over each value in the held-out row is lowest for CrossCat.
Figure 6-3: A comparison of missing value imputation accuracy between BayesDB and standard and sophisticated baselines: In this experiment, we took a 250 row, 100 column data table, removed a variable proportion of the data (x-axis): either 10%, 25%, 50%, 75%, or 90%, then loaded the data into BayesDB and generated 32 models with 300 iterations of analysis each, and used INFER to fill in all missing values with 500 samples from the posterior predictive for each missing value. CrossCat performs about as well as CRP Mixture at filling in missing data, but both are much better than Naive Bayes.

Figure 6-4: An empirical assessment of BayesDB’s calibration on synthetic datasets: These experiments were run on a synthetic data table with 500 rows, 4 columns, 2 views, and 8 categories per view, where each category’s data was generated from a multinomial distributed with 10 values. Afterwards, 20% of the data was randomly removed. The data was then loaded into BayesDB and analyzed with 32 models for 500 iterations. These three plots can be thought of as a multinomial P-P plot using normalized frequencies over missing values: each line represents the distribution of a column, where the y-axis is the inferred distribution and the x-axis is the actual distribution. These plots demonstrate that the three model types currently implemented in BayesDB (CrossCat, CRP mixture, and Naive Bayes) are all well calibrated.
Figure 6-5: Experimental evidence that BayesDB’s dependency detection is robust to the multiple comparisons problem: This experiment was run on a synthetic dataset with 100 rows and 32 or 64 columns, with two 2-column views. Therefore, the ground truth is that there are two pairs of columns that are dependent on each other, and all other pairs of columns are independent. This plot shows the dependence probability, as estimated by BayesDB using CrossCat, for all pairs of columns as a function of iterations. Each pair of columns is shown as a blue line, except for the two pairs of columns that are dependent according to ground truth. Before any iterations of inference, all pairs of columns start out with a fairly low yet uncertain dependence probability, accurately capturing BayesDB’s uncertainty. After a number of iterations, the dependence probabilities have converged to their true values, showing that BayesDB is able to discover individual relationships between variables in a robust way.
Figure 6-6: **Empirical validation of CrossCat’s MCMC sampler using the Geweke test:** The Geweke test is an inference diagnostic tool that compares the distributions of forward and backward sampled hyperparameters: in a correct sampler, the KL Divergence between those two distributions should converge to 0. The left plot shows a passing Geweke test, where all KL Divergences approach 0, and the right plot shows a failing Geweke test, where the KL Divergences do not all converge to 0 (this plot was used to diagnose a real bug in the sampler).
Figure 6-7: Estimates stabilize for DEPENDENCE PROBABILITY and INFER with more iterations. These plots show results on a real dataset: the Dartmouth Atlas of Healthcare, with 20% of the data randomly removed. The error bars shown in the plots are standard errors calculated over 5 runs, where each run has 20 models and 100 iterations of MCMC. In the top plot, the two lines represent the dependence probability between two pairs of columns: as the number of iterations increases, the estimates stabilize, and the probabilities move from 50% towards their true value. The bottom plot shows the proportion of the missing data that we are able to fill in with a confidence level of 10%: initially, only a small portion of the data is filled in, and the number of cells BayesDB is able to fill in at a given level of confidence increases with the number of iterations.
6.3 Inference Performance

CrossCat is memory efficient and the runtime of its inference algorithms scales linearly with the size of the data \((ND, \text{ for } N \text{ rows and } D \text{ columns})\) in practice (Figure 6-8) even though the worst case performance of these inference algorithms is \(O(ND \max(N, D))\) in theory.

That said, we have not yet shown that it still works well in practice beyond the 1M-cell tables we've been working on, or that the inferences are reliable enough at that scale, especially with large numbers of columns. At billions or trillions of cells, a different (and maybe more heterogeneous) inference backend might be more appropriate. However, there are some regimes (e.g. \(N < 100, D \approx 1000\), with sparse observations, or \(N \approx 1000, D \approx 10-100\), with lots of highly nonlinear, stochastic relationships) where the CrossCat backend might offer an advantage over many custom models.
Figure 6-8: **Timing Plots:** Inference time (time per step, the y-axis in these plots) increases linearly in both the number of rows and number of columns.
Chapter 7

Case Studies

Versions of BayesDB (and CrossCat) have been used to analyze a wide variety of datasets including surveys, performance metrics for hospitals, branches of nonprofits, econometric tables, feature vectors extracted from text/images, voting records, and gene expression microarrays. BayesDB can perform exploratory analysis, predictive modeling, and even confirmatory analyses by framing hypothesis tests as queries over the posterior over full joint distributions. This chapter walks through some of these case studies (for an additional case study, see Section 1.3).

7.1 Web Server Log

Another application domain that BayesDB is particularly well suited for is web browsing analysis. One MIT fraternity gave us an anonymized sample of their website’s Apache server logs, where each log entry was a single click. The data was preprocessed by combining clicks from the same user (with the appropriate referrer_id) into a single aggregate browsing session. Each row is a complete browsing session, and each column is an indicator function for whether certain events occurred during the session, e.g. page visits, form submissions, and error pages. The final data table contains 2992 rows and 31 columns.

The website contains general information about the fraternity, but the primary purpose of the website during Spring semester is to attract potential summer residents. The website has an informational marketing page about what it is like to live in the fraternity over the summer, and the desired action is for the user to click through to submit a form to apply. Therefore, the most interesting questions that need to be answered about the data are: what can the fraternity do to discover what makes people apply for summer housing, and what changes should it make to improve the application rate?
The first step is to upload the data:

```
bql> CREATE BTABLE server FROM server.csv;
+--------------------------------------------------------------+
| ip   | server_error | sent_post | session_time | ... |
+--------------------------------------------------------------+
| multinomial | multinomial | multinomial | continuous | ... |
```

And update the datatypes with `UPDATE SCHEMA` if any of the datatypes were incorrect. For example, if we wanted to model `session_time` as multinomial, we could:

```
bql> UPDATE SCHEMA FOR server SET session_time=multinomial;
+--------------------------------------------------------------+
| ip   | server_error | sent_post | session_time | ... |
+--------------------------------------------------------------+
| multinomial | multinomial | multinomial | multinomial | multinomial | ...
```

74
As in the previous case study, a good first step towards understanding a brand new dataset is often looking at the pairwise dependence probability matrix:

```
bql> ESTIMATE PAIRWISE DEPENDENCE PROBABILITY FROM server;
```

![Pairwise column dependence probability for server](image)

From this plot, we see that many variables in the dataset have a reasonable chance of being dependent on each other, but there are two strong clusters of dependencies. The large upper-left cluster is a group of variables showing whether a user visited each individual informational page. Additionally, the variable "/summersuccess.php" indicates whether the user successfully submitted a summer housing application form. It would be reasonable to infer from this plot that users who visit one informational page are more likely to visit another, and that the users who visited informational pages were more likely to apply for summer housing.

The large middle cluster of dependent variables contains information about the type of user: whether the
user is on an MIT IP address, what kind of browser was used, etc. It turns out that this fraternity website gets a large amount of traffic from bots, which can all be easily recognized because they try accessing random pages that do not exist, they do not have MIT IP addresses, and they use nonstandard browsers. Identifying this group of variables that is good at detecting the three types of users - fraternity members, bots, and other (all prospective summer residents would be in the "other" category) - allows us to smartly exclude bots from our future analyses.

We can also determine which variables are most predictive of whether the user applied for summer housing with mutual information. Dependence probability is extremely helpful to determine whether a relationship exists, but the way to quantify the predictive strength of that relationship is with mutual information.

```
bql> ESTIMATE COLUMNS FROM server ORDER BY MUTUAL INFORMATION WITH /summersuccess.php DESC LIMIT 9;
```

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>/summersuccess.php</td>
<td>sent_post</td>
<td>/summerapply.php</td>
<td>/summer.php</td>
<td>session_time</td>
<td></td>
</tr>
<tr>
<td>/summervirtual.php</td>
<td>/floorplan.swf</td>
<td>browser</td>
<td>/gallery.php</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The most predictive variables of whether a user applies for summer housing are sent_post (whether the user submitted a POST request - which is deterministically true for all users who applied for housing), /summerapply.php (whether the user visited the application page), /summer.php (whether the user visited the main summer homepage), and session_time (the longer the user spent browsing the site, the more likely it is that the user applied).
To get an intuitive sense of how long people typically spend on the website before applying, we can use `SUMMARIZE`:

```
bql> SUMMARIZE SELECT session_time FROM server WHERE /summersuccess.php = 'True'
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>session_time</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>continuous</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>19.0</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>5209.57894737</td>
<td></td>
</tr>
<tr>
<td>std</td>
<td>19955.2953493</td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>210.0</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>474.0</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>1055.5</td>
<td></td>
</tr>
<tr>
<td>max</td>
<td>87568.0</td>
<td></td>
</tr>
<tr>
<td>mode1</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>mode2</td>
<td>508.0</td>
<td></td>
</tr>
<tr>
<td>mode3</td>
<td>353.0</td>
<td></td>
</tr>
<tr>
<td>mode4</td>
<td>322.0</td>
<td></td>
</tr>
<tr>
<td>mode5</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>prob_mode1</td>
<td>0.105263157895</td>
<td></td>
</tr>
<tr>
<td>prob_mode2</td>
<td>0.0526315789474</td>
<td></td>
</tr>
<tr>
<td>prob_mode3</td>
<td>0.0526315789474</td>
<td></td>
</tr>
<tr>
<td>prob_mode4</td>
<td>0.0526315789474</td>
<td></td>
</tr>
<tr>
<td>prob_mode5</td>
<td>0.0526315789474</td>
<td></td>
</tr>
</tbody>
</table>

19 people applied for summer housing, and only two of them had 0-second browsing sessions. 0-second browsing sessions indicate that the person who submitted the form is most likely a bot.
We can get a better picture of how the few variables that are most predictive of applying relate by using the PLOT command, and by filtering out users who spent 0 seconds (they visited one page, then left) or excessively long (3000 seconds or more):

```
bql> PLOT SELECT /summersuccess.php, sent_post, /summerapply.php, summer.php, session_time FROM server WHERE session_time > 0 AND session_time < 3000;
```
BayesDB also has a way to examine a very interesting set of users: those who did not apply for summer residency, but were the most likely to apply given all the other information known about them:

```
bql> SELECT /summer.php, session_time, PREDICTIVE PROBABILITY OF /summersuccess.php FROM server WHERE /summersuccess.php = 'False' ORDER BY PREDICTIVE PROBABILITY OF /summersuccess.php LIMIT 10;
```

<table>
<thead>
<tr>
<th>/summer.php</th>
<th>session_time</th>
<th>PREDICTIVE PROBABILITY OF /summersuccess.php</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>470.0</td>
<td>0.841179679022</td>
</tr>
<tr>
<td>True</td>
<td>321.0</td>
<td>0.844736084322</td>
</tr>
<tr>
<td>False</td>
<td>90618.0</td>
<td>0.848060483723</td>
</tr>
<tr>
<td>False</td>
<td>178632.0</td>
<td>0.848133995579</td>
</tr>
<tr>
<td>False</td>
<td>395588.0</td>
<td>0.848390787098</td>
</tr>
<tr>
<td>False</td>
<td>18630.0</td>
<td>0.849089493422</td>
</tr>
<tr>
<td>True</td>
<td>533.0</td>
<td>0.851747458781</td>
</tr>
<tr>
<td>True</td>
<td>1454.0</td>
<td>0.851911474295</td>
</tr>
<tr>
<td>True</td>
<td>223.0</td>
<td>0.8561917774287</td>
</tr>
<tr>
<td>True</td>
<td>143.0</td>
<td>0.85859210474</td>
</tr>
</tbody>
</table>

This query reveals that the biggest missed opportunities for possible summer applications come from two types of users. The first user type visited the summer informational page and browsed for a few minutes (usually 2-8, but up to a half hour) but never applied. The second type of user spent an extremely long time on the site - on the order of 3 days - which could mean a tab might have been open for a few days before the user returned to the website. The fraternity website administrators could consider modifying their website in a way to appeal to these two user segments.

### 7.2 Illustrations on Real Data

To convey a sense of the wide variety of datasets that BayesDB is an appropriate tool for, this section will briefly outline three datasets that could be analyzed with BQL.

#### 7.2.1 General Social Survey

The General Social Survey is a large-scale demographic and attitude survey of U.S. residents. We loaded a subsample of 1975 rows (people) and 100 columns into BayesDB for analysis. We again start the analysis by examining the dependence structure:
There are some clusters of columns that are dependent with high probability. Looking closely at the clusters, we see that one cluster is about people's attitudes towards science and one is about banning books and other extreme conservative views. We could zoom in on these clusters by using `ESTIMATE PAIRWISE DEPENDENCE PROBABILITY FROM gss SAVE CONNECTED COMPONENTS WITH THRESHOLD 0.8 AS clusters;` then examine each cluster individually with `PLOT SELECT clusters.1 FROM gss`.

In contrast, let's see what we would find if we used correlation, a traditional statistical technique, instead of dependence probability:
In contrast to dependence probability, where it is easy to find clusters of possibly related columns, correlation is only able to capture the magnitude of linear relationships, which severely limits its usefulness for finding underlying structure in data. In the above pairwise correlation matrix, it is hard to see more than a few random dependencies between columns.

Now, we can pick a particular cluster of columns from the pairwise dependence probability matrix to examine further. One cluster shows relationships between how often they attend religious services, whether they obey their parents well, strength of religious affiliation (reliten: 1 means very religious, 4 means not religious) and whether they will allow anti-religious books in the library (libath: 1 means they will not allow anti-religious books, and 2 means anti-religious books are ok). To examine the nature of these relationships, we can use SIMULATE to generate new probable people conditioned on whether they will allow anti-religious books in the library or not.
From the above histograms, we can see that BayesDB learned that people who do not want to allow anti-religious books in libraries (the top plot) have stronger religious beliefs (a much higher proportion of them have `reliten == 1`).
7.2.2 Chicago Crime

Another dataset of interest is all crimes that occurred in Chicago in 2013. We used a tiny subsample with 501 rows (crimes) and 21 columns. One sample question that may be investigated in this dataset is “what factors about a crime influence whether an arrest occurred?”

```
bql> ESTIMATE PAIRWISE DEPENDENCE PROBABILITY FROM chicago_demo;
```

We can immediately see that arrest (whether an arrest was made) is dependent on time, domestic, iucr, primary type, description, and fbi code. Some are obvious - the FBI code contains information about whether an arrest was made - but it would be interesting to more closely examine the relationship between domestic and arrest.
BayesDB could also be used to perform predictive policing with SIMULATE, which can be used to generate new probable crimes, conditioned on a particular beat, for example:

```
bql> SIMULATE arrest, latitude, longitude, beat FROM chicago_demo GIVEN beat=1112 TIMES 5;
```

```
+--------------------------+-------------------------------------+--------------------------+---------------------+---------------------+---------------------+
| arrest | latitude | longitude | beat | arrest | latitude | longitude | beat |
+--------------------------+-------------------------------------+--------------------------+---------------------+--------------------------+---------------------+---------------------+
| false | 41.9206725829 | -87.7299463364 | 1112 | true | 41.9089931262 | -87.6976744089 | 1112 |
| true | 41.8829639368 | -87.7101279734 | 1112 | false | 41.8927956345 | -87.7101515023 | 1112 |
| true | 41.8878548392 | -87.717937836 | 1112 |
+--------------------------+-------------------------------------+--------------------------+---------------------+--------------------------+---------------------+---------------------+
```

BayesDB is able to simulate probable crime locations, and whether arrests are likely to occur at those locations, which could be used by a police department to determine where to position its patrols.
7.2.3 Flights

We obtained a dataset of origin, destination, and other itinerary details of 10% of all passengers transported from 1993 to 2013 from the Research and Innovative Technology Administration Bureau of Transportation Statistics, then subsampled the data further down to 21,000 rows and 17 columns before loading it into BayesDB.

```
bql> ESTIMATE PAIRWISE DEPENDENCE PROBABILITY FROM flights;
```

Many interesting trends are present in this dataset, including how ticket prices change as a function of which legs are in a trip, time of year, and airport.
For example, we could investigate which flights have the most unusual fare prices, given everything else we know about the flight.

bql> SELECT originairportid, rpcarrier, milesflown, itinfare FROM fl_tkts ORDER BY PREDICTIVE PROBABILITY OF itinfare ASCENDING LIMIT 9;

+----------------------------- -+--------------- -------------------- +----------------
| I row-id | originairportid | rpcarrier | milesflown | itinfare |
| -------------------------------------- -------------------- +----------------
| 1 16485 | 14057 | AS | 762.0 | 99738.0 |
| 2 5010 | 11618 | UA | 9806.0 | 3795.0 |
| 3 14754 | 13930 | US | 8469.0 | 3987.0 |
| 4 17103 | 12892 | AA | 4950.0 | 4975.0 |
| 5 19875 | 10372 | DD | 8256.0 | 4705.0 |
| 6 3997 | 14771 | AA | 5744.0 | 4556.0 |
| 7 7266 | 12266 | CO | 7839.0 | 3989.0 |
| 8 17403 | 13487 | DL | 2675.0 | 2184.0 |
| 9 5269 | 14679 | UA | 109.0 | 1495.0 |

The most unlikely fare was a fare of $99738, which is certainly an anomaly or outlier, especially given how short the flight was. The other fares with low predictive probability were also extremely high, and seemed to not depend too much on miles flown, carrier, or origin airport.
Chapter 8

Related Work

There are two branches of related work that BayesDB seeks to unite and improve upon: probabilistic databases, which are databases where relations, tuples, and values can be associated with probabilities instead of being certain, and automated machine learning systems, which aim to solve data analysis problems automatically, with little or no user input.

There have also been a number of attempts to put machine learning inside of databases, but these approaches are only tangentially related to BayesDB since their only goal is to allow users to run traditional machine learning algorithms, like random forests, on their data without having to export it out of the database [18] [12].

8.1 Probabilistic Databases (PDBMSes)

Many probabilistic databases have been introduced in the last ten years: there have been many attempts to store uncertainty in databases and introduce SQL-like queries on that uncertain data, such as Mauve and BBQ [43] [31][2] [10] [42] [19]. Almost always, though, the approach behind these probabilistic databases is to just learn a single model of the data instead of storing a set of many models \{\Theta\}, which allows predictive queries to average over all possible \Theta s [14] [40] [4].

In these approaches, the underlying inference engine tries to estimate the parameters of a graphical model (usually a Bayesian network or factor graph). Additionally, no existing probabilistic database supports graphical model structure learning (MC DB could have, but does not) because it is intractable to learn good graphical models for arbitrary data tables, even if the data is fully observed. While some success can be had with such an approach, limiting oneself to a single model of the data severely limits the flexibility of the types of queries that can be evaluated and limits the accuracy of answers to those queries.

For example, consider a version of BayesDB that was implemented with a single model, \Theta, which could
be sampled from \( p(\Theta|X) \) or the MAP \( \max_{\Theta} p(\Theta|X) \), instead of a set of models \( \Theta \) sampled from \( p(\Theta|X) \). Regardless of whether \( \Theta \) was found using CrossCat or graphical model structure learning, a single \( \Theta \) asserts a fixed dependence structure: under \( \Theta \), the probability that two variables are dependent is either 0 or 1, since there is only a single model. The way that BayesDB is able to answer queries like dependence probability with a probability instead of a 0 or 1 is because it averages over all \( \Theta \)s. Additionally, it could be possible that \( p(\Theta|x) \) is a very wide, flat distribution, in which case it is extremely hard to summarize with a single point estimate.

In addition to storing an entire collection of models instead of just one, BayesDB’s other advantage is its flexible query language, BQL. Most PDBMSes have query languages that look like SQL but with a couple probabilistic functions added, often as user defined functions (UDFs). BayesDB offers a variety of new first-class query primitives, including INFER, SIMULATE, ESTIMATE COLUMNS, and ESTIMATE PAIRWISE, which all answer statistical questions that are either impossible or awkward to express in existing query languages.

### 8.2 Automated Machine Learning Systems

With the popularization of “data science” in the past 2-3 years has come a wide variety of systems that attempt to make machine learning either automatic or easier.

There are a number of commercial attempts to develop a system that solves prediction problems automatically, such as DataRobot, PredictoBot, and DataSight, but their approach is always to try a variety of parameter settings (using grid search) for many predictive models (e.g. random forests, lasso regression, ridge regression, or SVMs) and selecting a model based on cross-validation accuracy. While that approach can work quite well to solve a specific prediction problem (predict \( x \) given \( Y \)), BayesDB is a fundamentally different approach: it learns models of the data that explain the entire dataset all at once, which once learned can answer predictive queries like “predict \( x \) given \( Y \)” or “predict \( x_2 \) given \( Y_2 \)” on the fly – in constant time, in addition to answering deeper queries that ask about the dependence structure of the data or detecting which data points are unlikely (low predictive probability).

There have been other more principled approaches to building an automated statistician, which build off the Bayesian nonparametric literature just as CrossCat does [5]. This particular approach focused narrowly on discovering structure in time series, and could be integrated into a streaming version of BayesDB. However no other attempts have focused yet on the tabular data regime, as BayesDB does.
Chapter 9

Conclusion

BayesDB has the potential to make basic data science accessible to many more people by allowing users with no statistics expertise to ask statistical questions of their data. Without specifying any algorithms or parameters, users receive results that avoid common statistical pitfalls such as the multiple comparisons problem and choosing inappropriate models.

9.1 Future Work

9.1.1 Software Engineering

Improved Query Optimization for SQL Queries

Typically, databases store histograms to approximate the marginal probability densities of each column, which are used during query planning to estimate the number of rows that will match some predicate. However, predicates in where clauses can sometimes have many conditions, e.g. WHERE a > 0 AND b = 0 AND c < 0 .... The database must try to estimate the number of rows remaining after applying all three predicates, yet it only has information on how restrictive each predicate is individually. With three predicates \( p_1, p_2, \) and \( p_3, \) it is reasonable to assume that each predicate is independent of the others, so the selectivities of each predicate, \( s(p_1), \) should multiply. However, in a dataset where \( p_1, p_2, \) and \( p_3 \) are strongly related, that could be a very bad approximation.

The obvious solution is to store a histogram of the entire joint probability of the data, but unfortunately, the size of the joint histogram scales exponentially in the number of variables. Instead, BayesDB’s models \( \{ \Theta \}, \) which are usually linear in the number of variables (e.g. for Naive Bayes, DP Mixture, and CrossCat), can be used to estimate the joint probability of the data to accurately estimate the selectivity of complex predicates in a space-efficient way.
StreamBayesDB

The current implementation of BayesDB does not support \texttt{INSERT}: when the data table is updated, it is necessary to recompute all models \{\Theta\}, because each individual \(\Theta_i\) is supposed to be an independent sample from \(p(\Theta|X)\), and now \(X' = X \cup x'\) after inserting \(x'\).

However, streaming workloads are increasingly common, so a natural extension to BayesDB is to develop an efficient \texttt{INSERT} command. For example, if the backend uses CrossCat, an approximation algorithm can be used to update \{\Theta\} to approximately include a new row without needing to re-run all of \texttt{analyze}: one such update could keep all views and all existing categorizations the same, but just sample a categorization for each view in \(x'\), and re-fit all component models appropriately.

The second, and more ambitious way that BayesDB could support streaming is by using a more complex probabilistic model: instead of each cell holding a single value, each cell could contain an entire time series of values. This enables an \texttt{UPDATE} command which takes a timestamp as an argument to be used to dynamically update the time series in a particular cell. This would allow time series to be modeled more naturally: currently in BayesDB, given a time series dataset, feature extraction must first be performed in order to generate rows that are summaries of independent buckets of time. While this approach to analyzing time series in BayesDB is effective, it is still slightly more awkward than it could be, optimally.

9.1.2 Performance Engineering

The current implementation of BayesDB has been tested on datasets with up to 1 million cells (rows x columns). The performance bottleneck in the current system is the CrossCat inference engine: the code that runs MCMC (Gibbs sampling, with a Metropolis-Hastings step in certain implementations) on CrossCat models \(\Theta\) in order to implement \texttt{analyze}. All other parts of the system, including the client, parser, BayesDB engine, and CrossCat prediction engine (that implements \texttt{predictive.sample}, \texttt{predictive.probability}, \texttt{column.dependence}, and \texttt{row.similarity}) are orders of magnitude faster.

However, it is likely that drastic speed improvements will be relatively straightforward to achieve, given that the current implementation of MCMC in CrossCat was not designed with performance in mind at all. Since large C++ objects are created and destroyed multiple times during a single MCMC transition, eliminating that overhead, along with cache optimization, should offer about a 10-100x speed up in serial runtime. Additionally, while CrossCat has been successfully deployed on Hadoop, the overhead involved with Hadoop frequently reading and writing to disk prevented any significant performance improvements, but a properly implemented distributed inference engine would allow within-chain parallelization, offering potentially another 10x improvement in runtime and scale.

Additionally, subsampling is a crucial yet simple future feature for BayesDB. With subsampling enabled, BayesDB initializes and analyzes its models from only a small fraction of the total number of rows in the
full dataset. Since initialize and analyze both run in time linear in the number of rows, this drastically reduces the amount of time it takes before the models are good samples from $p(\Theta|X')$, where $X'$ is the subsample from $X$. Recent work by Obermeyer, Glidden, and Jonas shows that subsampling in CrossCat is not only a powerful technique to get usable samples very quickly, but subsampling and then slowly adding more and more data back into $X$ is a technique that can be used to achieve faster convergence from the true posterior of the non-subsampled dataset, $p(\Theta|X)$ [26].

### 9.1.3 Modeling Improvements

While BayesDB and CrossCat already implement a number of useful datatypes, such as categorical, numerical, and experimental implementations of von Mises [1] (for cyclical data, e.g. time), it would be very useful and easy for an interested contributor to add new datatypes. Adding a new datatype is as simple as implementing the appropriate component model and likelihood function for the probability distribution of interest. Additionally, instead of requiring each new datatype to be implemented in Python and integrated into the existing codebase, making custom datatypes easy to add via probabilistic programming is an exciting possibility.

### 9.1.4 Data Analysis Applications

The most promising future work involves using BayesDB to quickly analyze a huge variety of datasets. The only requirement for a dataset to be a good fit for BayesDB is that each row needs to be able to be treated as a sample from some fixed underlying probabilistic process. However, even datasets that do not appear to fit into this format, such as time series data or a server log, can often be transformed into a BayesDB-friendly format with a small amount of data preprocessing and feature extraction.

A range of exploratory applied projects of this sort are beginning, including profiling of physiological growth, biomarker identification and analysis pipeline standardization for the Cancer Genome Atlas, and analysis of flight and travel datasets to build a service that provides travelers with trip recommendations.

### 9.2 Contributions

In summary, this thesis introduces BayesDB, a database that allows users without statistics expertise to query the probable implications of their tabular data with BQL. BQL provides model independence analogously to the way that the relational model provides physical data independence in SQL databases, which means that BQL allows data analysis tasks to be described independently of the models used to solve them. Contributions include the design of BQL, the architecture of BayesDB, and demonstrations of efficacy and scalability for BQL and BayesDB on both synthetic problems and several real-world case studies. We hope
BayesDB represents a step towards the development of tools for automated data analysis technology whose simplicity and flexibility approaches that of current tools for data management.
Bibliography


