

DBMS INNOVATIVE PROJECT

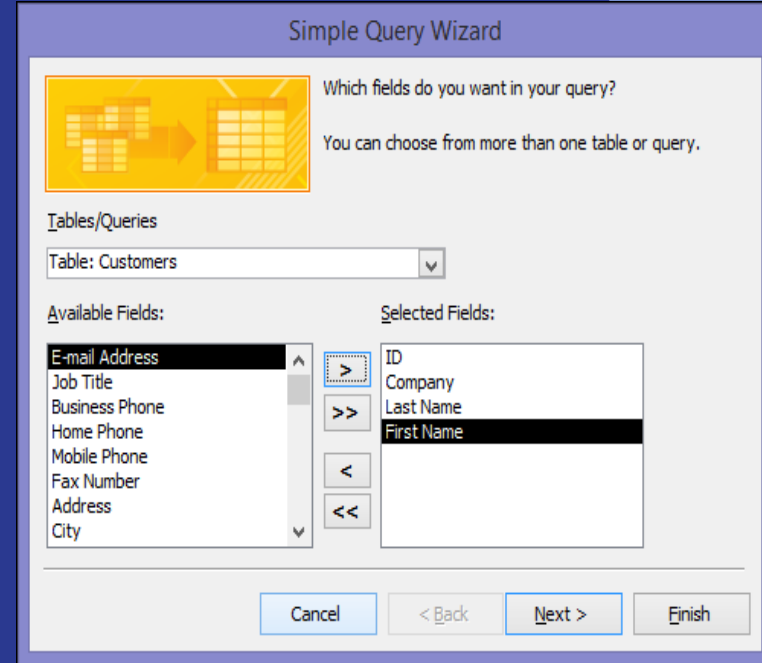
Dynamic Query Forms for Database Queries

Liang Tang, Tao Li, Yexi Jiang, and Zhiyuan Chen



INTRODUCTION

1. Modern scientific databases and web databases maintain large and heterogeneous data. Traditional predefined query forms are not able to satisfy various ad-hoc queries from users on those databases.
2. This paper proposes DQF, a novel database query form interface, which is able to dynamically generate query forms.
3. It helps in querying the Database even for non SQL programmers. We can write general queries and it will be automatically converted into SQL query.



The screenshot shows a 'Simple Query Wizard' dialog box. At the top, it asks 'Which fields do you want in your query?' and notes 'You can choose from more than one table or query.' Below this, there is a 'Tables/Queries' section with a dropdown menu showing 'Table: Customers'. The main area is divided into 'Available Fields:' and 'Selected Fields:'. The 'Available Fields:' list includes 'E-mail Address', 'Job Title', 'Business Phone', 'Home Phone', 'Mobile Phone', 'Fax Number', 'Address', and 'City'. The 'Selected Fields:' list includes 'ID', 'Company', 'Last Name', and 'First Name'. Navigation buttons include '< Back', 'Next >', 'Cancel', and 'Finish'. An icon of two tables with an arrow between them is shown in the top left of the wizard area.

METHODOLOGY

1. DQF aims to capture a user's preference and rank query form components. The generation of a query form is an iterative process and is guided by the user.
2. At each iteration, the system automatically generates ranking lists of form components and the user then adds the desired form components into the query form. The ranking of form components is based on the captured user preference.
3. F-measure is used for measuring the goodness of a query form. A probabilistic model is developed for estimating the goodness of a query form in DQF.

Query Form Enrichment	<ol style="list-style-type: none">1) DQF recommends a ranked list of query form components to the user.2) The user selects the desired form components into the current query form.
Query Execution	<ol style="list-style-type: none">1) The user fills out the current query form and submit a query.2) DQF executes the query and shows the results.3) The user provides the feedback about the query results.

APPROACH

A query form \mathbf{F} is defined as a tuple $(\mathbf{A}_F, \mathbf{R}_F, \sigma_F, \text{Natural Join } (\mathbf{R}_F))$, which represents a database query template as follows:

$$F = (\text{SELECT } A_1, A_2, \dots, A_k \\ \text{FROM } \bowtie (\mathcal{R}_F) \text{ WHERE } \sigma_F),$$

The user enters the queries **not in SQL format** but as in **general queries**, all the queries that user enters is converted into the form shown above using **splitting function**, all the tables and columns required according to users query is joined using **Natural join** and the corresponding results are shown based on the best guess of what users want.

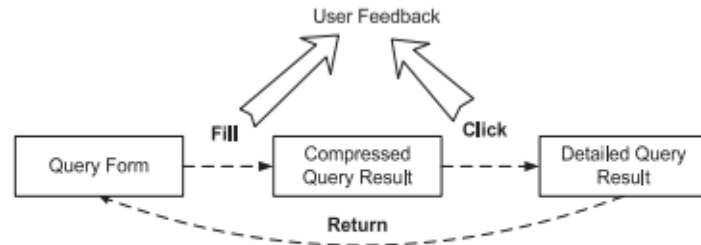


Fig. 2. User actions.

F	query form
\mathcal{R}_F	set of relations involved in F
\mathcal{A}	set of all attributes in $\bowtie (\mathcal{R}_F)$
\mathcal{A}_F	set of projection attributes of query form F
$\mathcal{A}_r(F)$	set of relevant attributes of query form F
σ_F	set of selection expressions of query form F
\mathcal{OP}	set of relational operators in selection
d	data instance in $\bowtie (\mathcal{R}_F)$
D	the collection of data instances in $\bowtie (\mathcal{R}_F)$
N	number of data instances in D
$d_{\mathcal{A}_1}$	data instance d projected on attribute set \mathcal{A}_1
$D_{\mathcal{A}_1}$	set of unique values D projected on attribute set \mathcal{A}_1
Q	database query
D_Q	results of Q
D_{uf}	user feedback as clicked instances in D_Q
α	fraction of instances desired by users

Concepts from Paper

Query forms are designed to return the user's desired result. There are 2 traditional measures to evaluate the quality of the query results, **precision** and **recall**.

Metrics: We now describe the two measures *expected precision* and *expected recall* for query forms.

Definition 2. Given a set of projection attributes \mathcal{A} and a universe of selection expressions σ , the expected precision and expected recall of a query form $F=(\mathcal{A}_F, \mathcal{R}_F, \sigma_F, \bowtie (\mathcal{R}_F))$ are $Precision_E(F)$ and $Recall_E(F)$ respectively, i.e.,

$$Precision_E(F) = \frac{\sum_{d \in D_{\mathcal{A}_F}} P_u(d_{\mathcal{A}_F}) P(d_{\mathcal{A}_F}) P(\sigma_F | d) N}{\sum_{d \in D_{\mathcal{A}_F}} P(d_{\mathcal{A}_F}) P(\sigma_F | d) N}, \quad (1)$$

$$Recall_E(F) = \frac{\sum_{d \in D_{\mathcal{A}_F}} P_u(d_{\mathcal{A}_F}) P(d_{\mathcal{A}_F}) P(\sigma_F | d) N}{\alpha N}, \quad (2)$$

where $\mathcal{A}_F \subseteq \mathcal{A}$, $\sigma_F \in \sigma$, and α is the fraction of instances desired by the user, i.e., $\alpha = \sum_{d \in D} P_u(d) P(d)$.

Expected precision is the expected proportion of the query results which are interested by the current user. Expected recall is the expected proportion of user interested data instances which are returned by the current query form.

Now we only need to estimate $P_u(d_{\mathcal{A}_{F_{i+1}}})$. As for the projection components, we have:

$$\begin{aligned} P_u(d_{\mathcal{A}_{F_{i+1}}}) &= P_u(d_{\mathcal{A}_1}, \dots, d_{\mathcal{A}_j}, d_{\mathcal{A}_{j+1}}) \\ &= P_u(d_{\mathcal{A}_{j+1}} | d_{\mathcal{A}_1}) P_u(d_{\mathcal{A}_1}). \end{aligned} \quad (4)$$

$P_u(d_{\mathcal{A}_{F_i}})$ in Eq.(4) can be estimated by the user's click-through on results of F_i . The click-through $D_{uf} \subseteq D$ is a set of data instances which are clicked by the user in previous query results. We apply *kernel density estimation* method to estimate $P_u(d_{\mathcal{A}_{F_i}})$. Each $d_b \in D_{uf}$ represents a Gaussian distribution of the user's interest. Then,

$$P_u(d_{\mathcal{A}_{F_i}}) = \frac{1}{|D_{uf}|} \sum_{x \in D_{uf}} \frac{1}{\sqrt{2\pi}\sigma^2} \exp\left(-\frac{d(d_{\mathcal{A}_{F_i}}, x_{\mathcal{A}_{F_i}})^2}{2\sigma^2}\right),$$

Definition 3. Given a set of projection attributes \mathcal{A} and an universe of selection expressions σ , the expected F-Measure of a query form $F=(\mathcal{A}_F, \mathcal{R}_F, \sigma_F, \bowtie (\mathcal{R}_F))$ is $FScore_E(F)$, i.e.,

$$\begin{aligned} FScore_E(F) &= \frac{(1 + \beta^2) \cdot Precision_E(F) \cdot Recall_E(F)}{\beta^2 \cdot Precision_E(F) + Recall_E(F)}. \end{aligned}$$

$FScore_E(F_{i+1})$ is the estimated goodness of the next query form F_{i+1} . Since we aim to maximize the goodness of the next query form, the form components are ranked in descending order of $FScore_E(F_{i+1})$. In the next section, we will discuss how to compute the $FScore_E(F_{i+1})$ for a specific form component.

IMPLEMENTATION

```
<body class="text-center">
  <div class="container">
    <h1>Database Management Systems MTE Project</h1><br><br><br><br><br><br>

    <form class="form-signin" method="POST" action="/">
      
      <br><br>
      <h1 class="h3 mb-3 text-white font-weight-normal">Dynamic Query Forms for
        Database Queries</h1><br><br>
      <input type="text" name="query" class="form-control top"
        placeholder="Type in your query here"required autofocus><br>

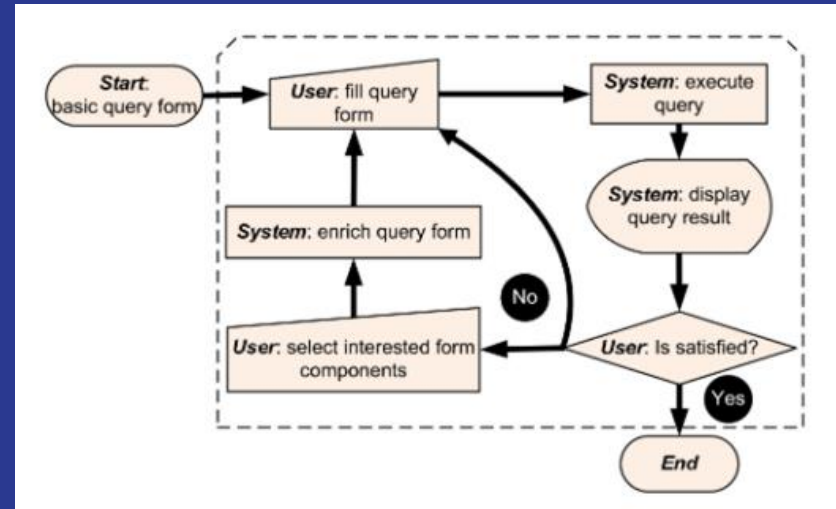
      <button class="btn btn-lg btn-primary btn-block" type="submit">Search in
        Database</button><br>

      <p class="mt-5 mb-3 text-muted text-white">&copy; Kunal Sharma and Kumar Apurva</p>
      <p class="mt-5 mb-3 text-muted text-white">Submitted To :- &nbsp;&nbsp;&nbsp;Prof. Goonjan Jain</p>
    </form>
```

1. First step is to setup a method from user to enter query forms in English language and parsing that to our server.
2. We hosted this projects's database on the phpMyAdmin Apache server, then we have created Node.js localhost server which acts as the user side.
3. All the SQL queries are handled using NPM package named mySQL, using the package we have connected the phpMyAdmin server to the client side and then we can execute queries from client side which gets transferred to backend server and are then executed.

The implementation includes the **User filling** in the query, then the query is parsed and the statement is divided into simpler words each containing some information about different columns from different tables, we created a function named **queryBuilder()** which recombines these words in such a way that it creates a valid SQL query from the generalized English query written by non SQL user.

After the display of the query User has an option to go with the query or re-select queries generated iteratively by the underlying **Feed-Forward Neural Network**, whenever a query that interests user is generated the User selects the query and the Web App understands what queries need to be generated when fed with specific keywords.



Programming Web App

Database Management Systems MTE Project



Dynamic Query Forms for Database Queries

Type in your query here

Search in Database

© Kunal Sharma and Kumar Apurva

Submitted To :- Prof. Goonjan Jain

- This is the query form generated on the client side, the user enter query in general English language and can access database from here. The client side is hosted on the localhost:3000 port.
- If the user wants to enter SQL queries or semi SQL queries then the Web App will detect what is expected from the query and will create the corresponding SQL query using Natural Join Operation on multiple tables.

- Since we have used **Express.js** so we called the function **require()** to import packages, we called SQL package from here and create an instance of it.
- Using **createConnection()** function we connected the Express and Node Server to the phpMyAdmin Apache Server where our database is hosted.
- Using **stringQuery.split()** function we divided the query into individual components and then reconstruct the SQL query corresponding to it using **queryBuilder()** function.

```
app.js                                     index.html
//jshint esversion:6
const express = require("express");
const bodyParser = require("body-parser");
const request = require("request");
const https = require("https");
//const mysql = require("mysql");

const app = express();
app.use(bodyParser.urlencoded({
  extended: true
}));
//for using static images, css files
app.use(express.static("public"));

app.get("/", function(req, res) {
  res.sendFile(__dirname + "/index.html");
  //console.log("SUCCESS HERE");
});

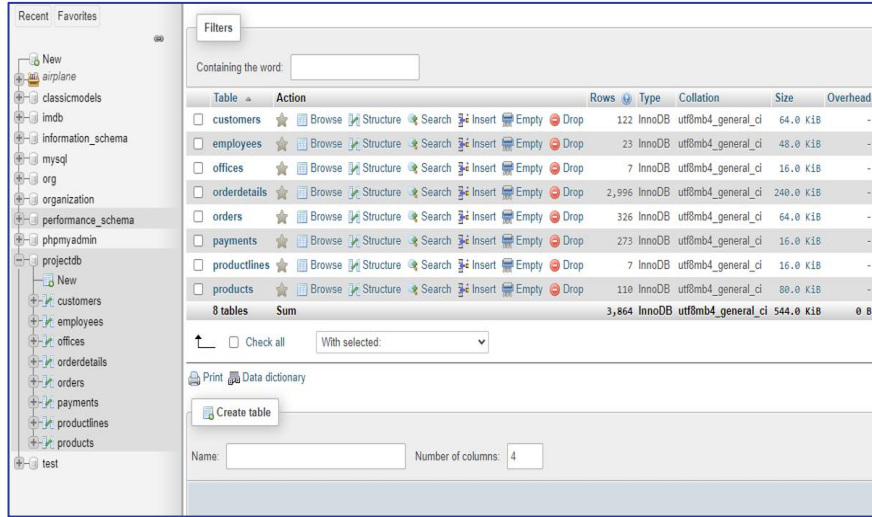
app.js
app.post("/", function(req, res) {
  const query = req.body.query;
  //console.log(query);

  let stringQuery = query;
  let wordsQuery = stringQuery.split(' ');

  for (let step = 0; step < wordsQuery.length; step++) {
    console.log("Query Number: ", step, wordsQuery[step]);
  }

  function querybuilder(num, wordsQuery = []) {
    var mysql = require('mysql');
    var connection = mysql.createConnection({
      host: 'localhost',
      port: 3200,
      user: 'root',
      password: '',
      database: 'projectdb'
    });
  }
});
```

Building the query using Natural Join



The screenshot shows a database management interface. On the left is a tree view of the database structure. The main panel displays a table list with columns: Table, Action, Rows, Type, Collation, Size, and Overhead. Below the table list are options to check all tables, a data dictionary, and a 'Create table' dialog.

Table	Action	Rows	Type	Collation	Size	Overhead
customers	⌕ Browse ⚙ Structure 🔍 Search ➕ Insert 🗑 Empty ⚡ Drop	122	InnoDB	utf8mb4_general_ci	64.0 KiB	-
employees	⌕ Browse ⚙ Structure 🔍 Search ➕ Insert 🗑 Empty ⚡ Drop	23	InnoDB	utf8mb4_general_ci	48.0 KiB	-
offices	⌕ Browse ⚙ Structure 🔍 Search ➕ Insert 🗑 Empty ⚡ Drop	7	InnoDB	utf8mb4_general_ci	16.0 KiB	-
orderdetails	⌕ Browse ⚙ Structure 🔍 Search ➕ Insert 🗑 Empty ⚡ Drop	2,996	InnoDB	utf8mb4_general_ci	240.0 KiB	-
orders	⌕ Browse ⚙ Structure 🔍 Search ➕ Insert 🗑 Empty ⚡ Drop	326	InnoDB	utf8mb4_general_ci	64.0 KiB	-
payments	⌕ Browse ⚙ Structure 🔍 Search ➕ Insert 🗑 Empty ⚡ Drop	273	InnoDB	utf8mb4_general_ci	16.0 KiB	-
productlines	⌕ Browse ⚙ Structure 🔍 Search ➕ Insert 🗑 Empty ⚡ Drop	7	InnoDB	utf8mb4_general_ci	16.0 KiB	-
products	⌕ Browse ⚙ Structure 🔍 Search ➕ Insert 🗑 Empty ⚡ Drop	110	InnoDB	utf8mb4_general_ci	80.0 KiB	-
8 tables	Sum	3,864	InnoDB	utf8mb4_general_ci	544.0 KiB	0 B

- The database we are working on is of a **Company** that supplies different products worldwide ranging from Cars, Motorcycles, Ships and Planes.
- There are 8 tables in the Database namely, Customers, Employees, Offices, OrderDetails, Payments, Product Lines and Products.

How we interpret queries ?

To get the non SQL queries into SQL queries, we used the technique of One-Hot Encoding, in which we first constructed a list of all possible keywords of interests that user can enter like products, prices, due Payments etc and added them in a dictionary.

Each time a user enters a query, those keywords from the dictionary are turned **Hot** (symbolised as 1) and others are remaining **Cold** (symbolised as 0) which are not in the query as entered by user.

Based upon the keywords that are turned Hot (or 1) and using the concept of **Natural Join**, an intermediate dataset is created in **JSON** format containing the most likely information selected from tables. This information is then sent back to user side from the Express Server as JSON data and can be easily viewed and copied from there.

```
function encode(phrase) {
  const phraseTokens = phrase.split(' ')
  const encodedPhrase = dictionary.map(word => phraseTokens.includes(word) ? 1 : 0)

  return encodedPhrase
}

const dictionary = ['customers', 'employees', 'offices', 'orderdetails', 'price', 'payments', 'products', 'productlines', 'productnames', 'buyPrice'];

//console.log(encode('Information about customers'));
let queryNumberGenerated = encode(req.body.query);
console.log("\nqueryNumberGenerated", queryNumberGenerated, typeof(queryNumberGenerated));

var queryNumberGeneratedArray = new Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
for (i = 0; i < queryNumberGenerated.length; i++) {
  queryNumberGeneratedArray[i] = queryNumberGenerated[i];
}
console.log("\nqueryNumberGeneratedArray", queryNumberGeneratedArray, typeof(queryNumberGeneratedArray));
```


Results

- To verify the results of the Web Application we wrote different queries related to the product and product lines tables expecting results from that table.
- The app understands the most important Attributes and automatically selects the corresponding Product information from different tables.

```
[
  {
    "productCode": "S10_1678",
    "productName": "1969 Harley Davidson Ultimate Chopper",
    "productLine": "Motorcycles",
    "productScale": "1:10",
    "productVendor": "Min Lin Diecast",
    "productDescription": "This replica features working kickstand, front suspension, precise scale and require special care and attention.",
    "quantityInStock": 7933,
    "buyPrice": 48.81,
    "MSRP": 95.7
  },
  {
    "productCode": "S10_1949",
    "productName": "1952 Alpine Renault 1300",
    "productLine": "Classic Cars",
    "productScale": "1:10",
    "productVendor": "Classic Metal Creations",
    "productDescription": "Turnable front wheels; steering function; detailed interior",
    "quantityInStock": 7305,
    "buyPrice": 98.58,
    "MSRP": 214.3
  }
]
```

- As we can see from the database snippet above, we can enter queries in SQL form, semi SQL form, or even general English sentence, we will always get the information from the Product and Product Lines table.
- The Web Application creates Natural Join by the important information extracts the relevant data from the table and show it in JSON format.
- We have tested the working on all the tables by entering complicated English statements but the App detects the relevant and important information automatically and projects the corresponding results.

productCode	productName	productLine	productScale
S10_1678	1969 Harley Davidson Ultimate Chopper	Motorcycles	1:10
S10_1949	1952 Alpine Renault 1300	Classic Cars	1:10
S10_2016	1996 Moto Guzzi 1100i	Motorcycles	1:10
S10_4698	2003 Harley-Davidson Eagle Drag Bike	Motorcycles	1:10
S10_4757	1972 Alfa Romeo GTA	Classic Cars	1:10
S10_4962	1962 LanciaA Delta 16V	Classic Cars	1:10
S12_1099	1968 Ford Mustang	Classic Cars	1:12
S12_1108	2001 Ferrari Enzo	Classic Cars	1:12
S12_1666	1958 Setra Bus	Trucks and Buses	1:12
S12_2823	2002 Suzuki XREO	Motorcycles	1:12
S12_3148	1969 Corvair Monza	Classic Cars	1:18
S12_3380	1968 Dodge Charger	Classic Cars	1:12
S12_3891	1969 Ford Falcon	Classic Cars	1:12

CONCLUSION AND LEARNING

1. We can conclude from the project that SQL queries can be generated from the Semi-SQL and English sentences given that we can reconstruct the meaning of the query using the concepts of Dynamic Query Forms for Database Queries.
2. We understood the relevance of the Expected Recall and Precision which tells us about how many queries returned by the user are the actual queries that User wants and what proportion of queries are not the actual queries of User interest.
3. The F-score or F-measure is a measure of a test's accuracy. It is calculated from the precision and recall of the test, where the precision is the number of true positive results divided by the number of all positive results, including those not identified correctly, and the recall is the number of true positive results divided by the number of all samples that should have been identified as positive. This score helps in maintaining the relevance of a particular search within Database.
4. We understood how we can use Express.js with Node.js to create a Server and also learned how we can integrate SQL database into phpMyAdmin Apache server for communication.
5. We learned how to approach and understand the Research papers related to DBMS and how we can implement the Database based Servers on LocalHost computer.

The background is a solid pink color. In the top right corner, there is a decorative arrangement of overlapping squares and triangles in various shades of pink, creating a geometric pattern.

Thanks Everyone