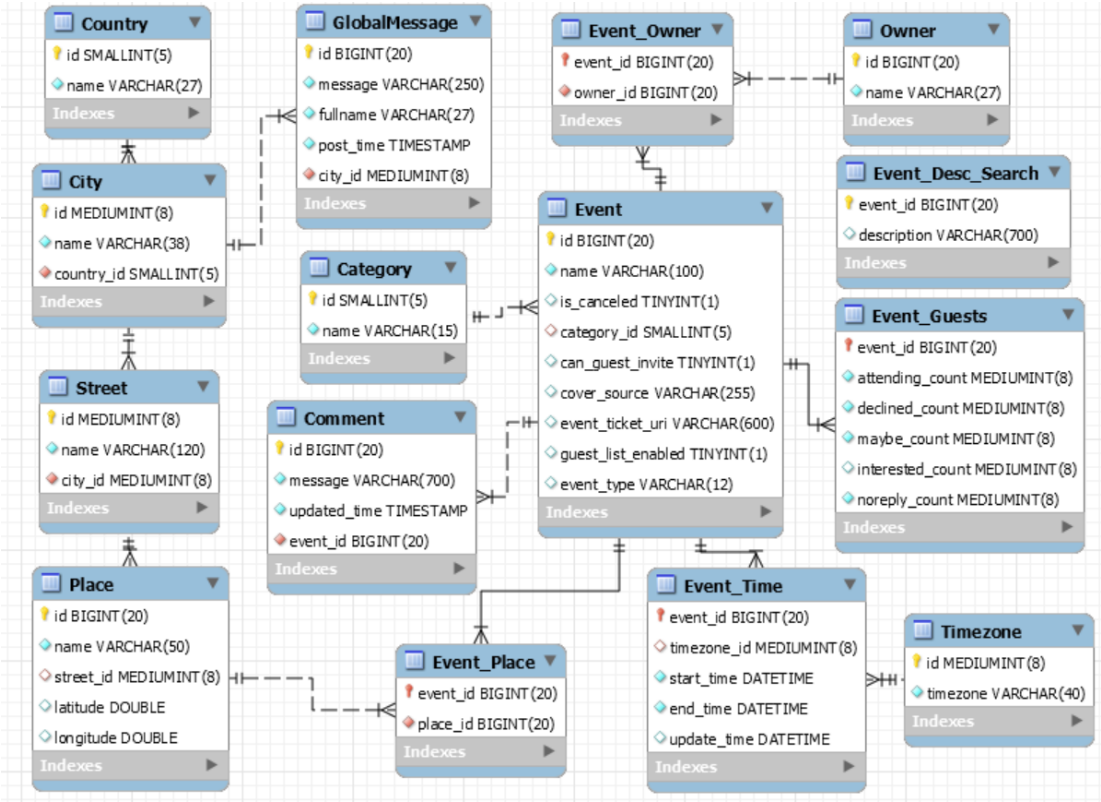
Village Mouse Documentation

Software Documentation

DB scheme structure



The Tables:

1. Event – a table that holds the basic information about an event, like its name and link to cover photo. This is the central entity we base our database around, so to make the database more efficient we aimed to keep this table as small as possible and join it with other tables where additional data is needed (all tables starting with the prefix Event below). All events are identified by a unique id.

   Primary key: id.

   Foreign Keys: category_id to "Category" table.

2. Category – a table that holds event's possible categories. This is a predefined enum that depends on Facebook's categories. We assign them unique ids to identify categories in a unique way in our system, and to make joins more efficient (better use a numeric key than a VARCHAR when joining).

   Primary key: id.

3. Comment – a table that holds comments about events. There are many comments in the system and they allow users to enter a large amount of text (700 characters). We join comments with the Events table when we want to show the comments associated with events.

   Primary key: id.

   Foreign Keys: event_id to "Event" table.

4. Owner – a table that holds information about owners of events. In the near future the system may store additional data about event owners (such as their personal details, contact number, preferences if they come back to the website, etc) so it makes sense to separate this data out and put it in its own table (we avoided pulling personal data about users from Facebook so this table is kept to a minimum).

   Primary key: id.

5. Event_Owner – an associative table that connects between Event and its Owner. The purpose of this table is to reduce the size of the Event table and break away additional attributes from the Events table (such as owner id column). We allow Join operations between the Owner & Event tables using this table in an efficient manner. A downside we had to consider to our method is that this table keeps foreign keys to Event & Owner tables, and since we can't keep foreign keys in the opposite direction as well, we can no longer maintain that Events have a corresponding owner in the Owners table. In a design perspective – we sacrifice this "integrity shield" to earn performance regarding the speed of our queries (a possible alternative that prefers to protect the database with the price of speed is to keep the owner id in the Event table and enforce it with a NON NULLABLE constraint).

   Note: later on we made some other design choices that preferred data integrity to speed (such as when choosing InnoDB as a database engine, see chapter about DB optimizations below).

   Primary key: event_id.

   Foreign Keys: event_id to "Event" Table

   owner_id to "Owner" table

6. Place – a table that holds information about places of venues hosting events, such as the Place's name and it's latitude and longitude coordinates. This table keeps a foreign key to Streets to be able to join the Place with the address of the venue (we allow a chain of joins to get more and more information about the location such as the city and country as well). We allow Places without streets in our database (marked with street_id = -1), and therefore we separate these two entities.

   Primary key: id.

   Foreign Keys: street_id to "Street" Table

7. Street – a table that hold information about streets where events in the system occur. We keep a foreign key to Citys to allow a fast chain of joins to get more information about the place where the event is occurring. Multiple streets may share the same name so this is another reason to include the city id in this table – as it enables to distinguish between them. We chose a dedicated id attribute as a primary key instead of using a combination of street name and city id for two reasons: keys composed of a single integer are preferred in terms of efficiency to a tuple where one of the attributes is a VARCHAR. The second reason is that theoretically in America 2 streets may share the same name in the same city! (Which is why Americans use zip codes, but we don't keep those in our database as we deemed them as redundant data).

   Primary key: id.

   Foreign Keys: city_id to "City" table.

8. City – a table that hold information about cities. Cities keep a foreign key to Countries, for the same reason described above in the Street table.

   Primary key: id.

   Foreign Keys: country_id to "Country" table.

9. Country – a table that hold information about countries (this is the highest form of labeled geo-location we can get about a place, which marks the end of a the place-street-city-country chain).

   Primary key: id.

10. Event_Place – an associative table that connects between Event and it's Place. This table exists for the same reason the Event_Owner table exists – to reduce the amount of columns we keep in the Event table and accelerate queries that SELECT from it (most queries do). Tables that need information about an Event's place may JOIN with this table, and may JOIN with additional Street, City, Country tables to get more info about the location of an Event..

    Primary key: event_id.

    Foreign Keys: event_id to "Event" Table

    Place_id to "Place" table

11. Timezone – a table that hold information about time zones in the world (logically important for users since worldwide event times differ by time zones).

    Primary key: id.

12. Event_Time – a sub class table that adds times based information to Events such as start_time, end_time, updated_time, etc. Not all Events keep time data! This is why this table serves as an optimization to decrease the size of the Event table – there is no point in keeping empty fields for some of the records. When event times are needed, Events may be joined with this table to fetch this data.

    Primary key: event_id.

    Foreign Keys: event_id to "Event" Table

    timezone_id to "Timezone" table

13. Event_Guests – another sub class table that holds information about the Event's guests, like it's attending_count, interested_count and so on. This table exists for a similar reason the Event_Time table exists, and it aims, of course, to reduce the size of the Event table (there is no point letting the db handle a big Event table if most queries don't use the guest statistics).

    Primary key: event_id.

    Foreign Keys: event_id to "Event" Table

14. Event_Desc_Search – a table that holds textual descriptions of events and used in order to execute a Full Text Search Query. This is the only table in the database that uses Myiasm database engine (as explained below). To allow Foreign keys in the rest of our tables using InnoDB engine, we had to break this one away to it's own table. Otherwise MySQL 5.5 complains (as Myiasm parses and ignores Foreign keys, MySQL does not allow Foreign keys between Myiasm and InnoDB tables). The FULLTEXT index on the description attribute was especially crucial to optimize our free text search queries that match words to contents of Event descriptions. There is another benefit to this table: Event description is the biggest column Events may hold, so keeping it separate helps reduce the size of the Event table records considerably.

    Primary key: event_id.

15. GlobalMessage – a table that hold messages of users of our application, like the name of the user and the message itself. This table keeps a foreign key to the Citys table to allow JOINing and fetching the geo-location of the person who commented.

    Primary key: id.

    Foreign Keys: city_id to "City" Table.

We used the CHECK syntax to maintain additional integrity on our tables (for example, don't allow Comment.updated_time that comes after the present time, or a negative attendants_count for Event_Guests, or a negative id for most tables).

Unfortunately MySql does not support CHECK constraints – it simply parses them and ignores them. We still left them in our database creation script as a good practice (as a side note: there is a bypass for creating CHECK constraints in MySQL by adding triggers on the DBMS, which we didn't use).

DB Optimization

We created indices for crucial fields in our database. Every field used for searching over the database in an execution of one of our queries is indexed.

Our indices are:

1. Index for 'country_id' field on City table.
2. Index for 'name' field on City table.
3. Index for 'city_id' field on Street table.
4. Index for 'street_id' field on Place table.
5. Index for 'category_id' field on Event table.
6. Full Text Index for 'description' field on Event_Desc_Search table.
7. Index for 'attending_count' field on Event_Guests.
8. Index for 'start_time' field on Event_Time table.
9. Index for 'event_id' field on Comment table.
10. Index for 'place_id' field on Event_Place table.
11. Index for 'owner_id' field on Event_Owner table.
12. Index for 'post_time' field on GlobalMessage table.

Also, indices for our table's primary key attributes were created automatically so we avoided creating additional redundant indices manually.

We also note that "InnoDB" Engine indexes Foreign keys automatically so we didn't have to index them manually. However, due to their importance in JOIN operations we still wanted to add a description of those indexes to our project documentation.

For our table's storage engines, we used "InnoDB" for all our tables except for the Event_Desc_Search table that supports Full Text Search queries by utilizing a FULLTEXT reverse index. FULLTEXT indices are not supported by InnoDB on MySQL 5.5 (only from MySQL 5.6 and on) so this table uses the "MyISAM" storage engine instead.

A point should be mentioned here about our design choices: we preferred data integrity to read / insert speed, which is why we chose InnoDB as the major database engine that most of our tables use. Foreign keys were crucial for us as we've decomposed our database to small tables, that maintain integrity by using Foreign keys. Indeed, Myiasm could have been beneficial in our case as well in terms of speed since most of our queries are "read" or "insert" queries (there is only one "update" query we use which would perform slower).

Following these lines, another optimization we used was splitting some tables into parts in order to keep small tables instead of one big table. For example, in our original database, the "Event" table contained fields about the event's owner, event's place, event's description, event's times and event's counts of guests. This table was, from our point of view, too big to be efficient as some records were redundant for certain events, and

definitely none of the queries required all this data to be selected at once. Therefore we split it into an "Event" Table and 5 more sub tables with foreign keys and indices of their own.

Lastly we made an effort to keep our table sizes to a minimum by setting correct data types and a fitting size limit for some types. For example: tables that hold medium amounts of data such as Country use MEDIUMINT as id, as opposed to Event and Comment which uses BIGINT for ids. For Boolean fields kept in the Event table we used TinyInt(1) as 1 bit of data.

DB Complex Queries

Output	Query Description	תיאור השאילתא
Event_name Event_cover Event_description	Mosaic Query : The newest event for the 8 categories that have the newest events (newest in start_time).	האירוע הכי חדש עבור 8 הקטגוריות בהן האירועים הכי חדשים (חדשים מבחינת זמן התחלה עתידי)
City_name Event_name Event_description	Hottest City – 10 events with the closest start date to the current date (in the future) in the city that has the highest number of events.	"העיר החמה" : מחזיר את 10 האירועים עם התאריך הקרוב ביותר ליום הנוכחי בעיר שבה יש הכי הרבה אירועים
Event_name City_name Event_description Event_start_time Event_end_time Event_attending_count	All events in the hottest season in the city hosting the biggest event: Returns all events with the most attendants in the city that hosts the event with the largest amount of attending people, and the city hosts at least 10 events total. The events returned must also occur within the same season as the event with the maximal amount of attendees (meaning 2 months before or after the maximal event, can occur on different years)	"העונה החמה בעיר עם האירוע הכי גדול" : מחזיר את האירועים עם הכי הרבה משתתפים מהעיר שבה יש את האירוע עם הכי הרבה משתתפים, ובעיר יש לפחות 10 אירועים בסה"כ. האירועים שיחזרו חייבים להתקיים באותה עונה כמו האירוע בעל המס' הגדול ביותר של המשתתפים: בטווח של חודשיים לפני או אחרי האירוע (אך יתכן שהאירועים מתקיימים בשנים שונות).
Category_name	The categories of the 10 events with the highest number of comments	הקטגוריות של עשרת האירועים בעלי מספר התגובות הגבוה ביותר
Owner_name Owner_event_attendings Number_of_comments	Most Popular Owners – 10 owners that their events has the highest number of comments together among the 20 owners that their events has the highest number of attendings together.	הבעלים הכי פופולארי – עשרת הבעלים שלאירועים שלהם יש בסך הכל את מספר התגובות הכי גבוה מבין עשרים הבעלים שלאירועים שלהם יש בסך הכל את מספר מאשרי ההגעה הכי גבוה.
Place_name Country	10 places that got the highest number of events from the	עשרת המקומות שבהם מתקיים המספר הגדול ביותר של אירועים

City Street Category_with_highest_attendings Number_of_events_from_category	category with the highest number of attendings.	מהקטגוריה שמספר מאשרי ההגעה לאירועים שלה הוא בסך הכל הגדול ביותר
Owner_name Number Of Events with the word 'Love'	Full Text Search Query – Most Sentimental Owners – 10 owners that got the highest number of events that has the word "Love" in their description.	**Full text search query** הבעלים הכי רגישים – עשרת הבעלים שהם בעלים של המספר הגדול ביותר של אירועים שבתיאור שלהם מופיעה המילה "Love".
Street.name City.name Country.name Number Of Events	Among all the streets in USA and UK, who are the 5 streets that got the highest number of events.	מבין כל הרחובות שבארה"ב ובבריטניה, מי אלה חמש הרחובות שבהם מתקיימים הכי הרבה אירועים.

Special Queries

1. COMPLEX Location query – we used the latitudes and longitude of a Place of Event in order to find the nearest events within a radius of 4 kms around the current event, and present them to the user using a Google Maps API. The query also returns the distance of each of the events from the center of the search circle.

2. INSERT Query – Add a new comment to a specific event. The comment will be inserted to the "Comment" table with an appropriate event_id (auto incremented, and “stamped” with NOW() updated_time).

3. INSERT Query – Send a "Message To The World" through our website. The message will be inserted to the "GlobalMessage" table. This is a context aware feature that lets the user choose a combination of a city and country aside from entering a name and message text. The final record inserted references the City table from the GlobalMessage table using a city_id column.

4. UPDATE Query – Using the API of facebook in order to update some of the guests counters of one specific event. The counters that will be updated are "interested_count" and "noreply_count". We chose not to update the "attendings_count" field because there is an index on this field and the update process would be slow because of the needed time to update the index (InnoDB has to sort the index field again due to the changed value).

   We also decided not to update "declined_count" and "maybe_count" because these Facebook Event fields can be read correctly only by using Facebook API’s User Token (these fields are fields of private Facebook events, and always return 0 for public events). If we will updated them, we will set both to 0 for any event because we utilize only a Facebook API App Token in our application, and so, in order to not lose data we decided not to update those fields.

   Note: The data we fetched initially from Facebook by using a User Token. This was necessary since searching for event ids in using Facebook’s Graph api requires a User Token (even for public events).

5. Input Query – Search event by it's id, when the user chooses an event, the query will be executed and return the event's details.
6. Input Query – Search events by input of a word in their description. This is a free text search query that fully utilizes the FULLTEXT reverse index on the description attribute of Event_Search_Desc table.
7. Input Query – Count Event By City – returns the number of event that occurs in the city input (content aware – the user chooses a city and gets in return the number of events occurring in it).

Query Optimization

As mentioned before, we created an index for every field that plays a part in a **search** process in at least one of our queries. These indices helped us to optimize the computation time of our queries because they store indxed fields in sophisticated data structures that support efficient search methods (for example through sorted data structures such as B-Trees used by InnoDB).

The FULLTEXT index greatly improved the speed of our “search for sentimental owners” query, as well as the “free text search query” as both needed to perform extensive search through text and perform a MATCH operation otherwise done sequentially.

We avoided running update queries on indexed columns as that would require the database to take more time since the sorted order needs to be maintained after update, which introduces a significant overhead.

We also avoided adding too many indices to the tables as that would obviously downgrade the overall performance of our queries.

We aimed to keep our tables as small as possible by defining restrictive data sizes for each column (it’s as big as our data from Facebook demands..).

Finally, we also mentioned that we split our tables, and so, because of the complexity of our queries we needed to execute quite a few join operations between tables, and because our new tables relatively smaller we were able to perform these joins operations quickly by using indexed Foreign Keys between smaller tables and so, according to our understanding we managed to improve our database performance in terms of computation time of our queries.

Code Structure & General Flow

Pre-Production Process

1. We created the DB scheme and all required fields and keys by running the CREATE TABLE part of **CREATE_DB_SCRIPT.mysql**. All data fields were set to encoding – unicode-utf8 to support multilingual text in the website.
2. Next we fetched the data we needed from facebook using our retrival script tool: **fb_extract.py**

The ran the script multiple times in *download_data* mode in order to download many Facebook Events & Comments using Facebook Graph's API. We used Json files to buffer what we store for each download batch. We had to execute the script multiple times due to Facebook's api limitations (can only search for events by keywords so we created a predefined keyword file with words of interest which we iterate and searched according to). After that we operated the script in *populate_db* mode which parses all the Json files we gathered (containing events and comments), separated them into logical units of data relevant to our database, and executed many many INSERT queries. We made sure to commit only every few comments to speedup the process but avoid a case where an error occurs and the script completely fails (note this is a bit different from the way the web-server functions as it runs queries in auto-commit mode).

Finally we created the indices from the 2nd part of **CREATE_DB_SCRIPT.mysql**, after the database was successfully populated.

From that point onwards we could start coding our project, performing ALTER operations on the database when bug fixes and performance tuning was needed.

The data retrieval tool resides in the API-DATA-RETRIVAL directory, and is ran in the following manner:

python fb_extract download_data *// Harvests selected events from facebook to local Json files*

python fb_extract populate_db *// Populates DB with data stored in Json files*

fb_extract uses 2 additional modules: fb_populate which contains the core logic of parsing Json files and inserting them into the database according to the tables structure, and fb_config which contains important constants used by the script (such as database connectivity details, and the categories enum mapping table which we need to keep track of the categories of the facebook event entities we parse).

Fb_extract has the following dependencies:
1) facebook-sdk version 2.00 (installed on Nova)
2) MySQLDb (installed on Nova)

The web-server

Our Flask's web-server resides in the Server.py file, and is run using:
python Server.py [port]
Where port is optional and set to 40666 by default.

This is where the majority of the queries are executed (with the exception of the update query which uses facebook and resides in *fb_update.py*). The methods in the web server closely correlate to the queries we execute and the pages we load in our web app.
A normal process starts with running Server.py, which waits for requests from this points onwards.
The server will start executing queries when users load pages.

Gui:
The gui is implemented Angular to connect the query results to the web page and Bootstrap to support a shiny look for the web-pages.

Index.html
Is the main page of the application, and we load multiple views for each tab in our menu (see them all inside the *views* folder). The *css* folder defines the look and feel of our application as bootstrap uses it. *Icons* and *img* contain image resources the web site use (big images for mosaic in main page and small category icons, as well as village mouse logo in the header). *Js* contains angular code hand written by us to initialize some widgets in our pages – for example the google maps initialization code resides here. This is the part that manages communication with the web-server's requests and responds. Finally *lib* contains some framework scripts of Bootstrap and Angular.

queries
This folder contains the queries we implemented in the project. All queries executed by the web-server reside in this folder as *sql* files and are parsed when a query is needed, and executed as prepared statements.

The normal flow is loading the *mosaic* view when a user first enters the web-site. Angular sends a request to the Flask webserver (listening to requests). Flask executes the "mosaic" query on MySQL DBMS ("select 8 newest events from the 8 categories with the newest events"). Flask receives the response from MySQL DMBS using a cursor (described below), and wraps the response in a neat Json sent back to the client. On the Client side Angular (mouse.js) parses the response and populates the Bootstraps components with the data that arrived from the query.

From this point on the user may click on the tabs, search for text or add comments. The usual pattern follows, where Angular is used to handle data backand forth in response to user events (load page, click button…) from the Flask web server. Server.py executes the queries on MySQL server (parsing the queries from the *quries* folder every time a query is needed) and returns the response to mouse.js which populates the user's "*view*".

- Excerpt on handling MySQLdb connectivity

Sine this is a databases course we repeat the connectivity process here:
In this project we used MySqlDB, in order to connect to DB and execute queries via Python, we have used, *'import MySQLdb',* to import python's wrapping library for MySQL.
We connect to the DB in the following way:

```
Con = MySQLdb.connect (database_hostname, 'DbMysql08', 'DbMysql08',
'DbMysql08', charset='utf8')
```

We used 'utf-8' coding in order to support Unicode characters in multiple languages (as Facebook returns data in Unicode).

The next stage is to create cursor for executing queries.

```
Cur = con.cursor (MySQLdb.cursors.DictCursor)
```

In order to prevent from auto commit, in order to be able handle transaction rollback by ourselves:

```
Con.autocommit (false)
```

The queries was executed in the following way:

```
Cur.execute([Query], [Parameters])
```

The parameters wasn't empty in insert or update. In order to commit the changes we used Con.Commit() or in case of failure Con.rollback(). We aimed to keep the data base in consistent and stable state. Any DB exception was handled and printed to log message without stack the program. The server runs in autocommit mode so MySQLDB takes care of rolling back on errors (speaking of errors: we have included a try catch on the main function of the server so it gets restarted when bad errors occur, to increase fault tolerance).

Description of API

- Facebook API Code

In order to communicate with facebook data base we used Facebook Graph API via python. The main python lib the implement the connection is '`import facebook`'.

API Usages

1. Create graph API object:
   ```
   graph = facebook.GraphAPI(access_token=app_token, version='2.2')
   ```
2. Get events by keyword:
   ```
   graph.request('search',{'access_token': user_token,'q': key,'type': 'event'})['data']
   ```
3. Get Event details by EventID:
   ```
   graph.get_objects(ids=[eventID], fields= [Required Fields])
   ```
4. Get All posts on specific event by event id:
   ```
   graph.get_connections(id=event_id, connection_name='feed')
   ```

All the data from facebook API in JSON format. In the next section JSON parsing will be explained. The data was reorganized in order to match to our DB scheme.

Facebook API Error Handling

We ignored facebook.GraphAPIError, in order to create stable facebook API usage without crashing due to third party exceptions. In addition, there are some facebook API restrictions for the using of API. If we exceed maximum retry count ConnectionError will be thrown. The code catch the exception, holds for 60 seconds and retry again. Any other exception is caught and handled in the following way: Log description is printed, and the program continues, without crashing.

Script flow:

1. Open Keyword.txt file and get all keywords. After it get Events ID's from Facebook DB using keywords.
2. For each Event ID get event details from Facebook DB in JSON format.
3. Save all events details to data.json file, using json.dump.
4. For each Event ID get all posts to this event as JSON format.
5. Save posts in (Event_ID)_comments.json file

External packages/libraries

1. Facebook-sdk – a python library wrapping Facebook's Graph 2.2 API for fetching data from facebook.
2. MySQLdb – a Python wrapper for handling MySQL databases via Python.
3. Flask – for the programming of our web server.
4. Angular – in order to connect between our website's GUI and the data that returns from our database.
5. Bootstrap – for the display of our website.

Our code runs using Python 2.7.

Bonus

- *Originality in design* – One decision we're proud of is breaking away the description table to Event_Desc_Search and create a search table, which was a major optimization for us. Hopefully the description in previous chapters makes justice with this optimization..
- *Interesting algorithms implemented* – see selection query by distance utilized by google maps.
- *Aesthetic Gui* – Check (mind the logo)!