# Design document

## Overview

Implemented in C#/.NET core . Instructions for compiling and running the program are in a separate document.

The program takes a batch.txt file as input containing one more more lines which have the file names of the json files storing the MongoDB query. The queries will be executed in the same order as the order in which files appear in the batch file.

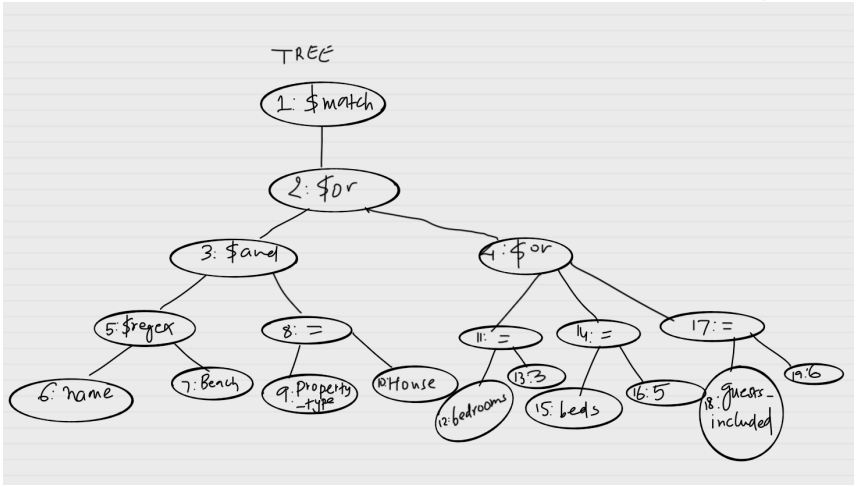
To show the improvement that query caching has, the program will print whether a query is a hit or miss, the time taken to execute the query, the no. of records returned, whether eviction took place, and if yes, which/how many items were evicted from the caches.

## Query parsing/translation

We parse the incoming json query into a expression tree and a set of projected attributes. As soon as a query is executed on MongoDB cloud, we store it in the query cache. To check if a subsequent query is covered by a cached query, we check if the cached query is identical to the new query or borader in scope than the new query. If yes, we consider it a hit. Otherwise, it's a miss.

## Flow

1. Parse the query. The logic resides in `MongoDBQueryCache.MongoDBQueryParser::ParseQuery()`. One important result of parsing is an expression tree. To keep things simple, we consider only a limited set of query operators such as `$and`, `$or`, `$regex`, etc.




2. Next, we check if the query is already satisified by an existing query cached locally. We load all the cached queries and see if we can find

one that satisfies (or "covers") the new query. This logic resides in `MongoDBQueryCache.Query::Covers()`.

a) the logic is not exhaustive enough to find satisfiability for any combination of queries, but has been designed to at least work well with the queries considered for this assignment. For example, it can identify correctly cases where the new query has two clauses connected by `$and`, and the cached query has the same two clauses connected by `$or`.

b) one simple but important check we make is whether the cached query had at least all the attributes projected that the new query requests for.

c) the key idea we rely on to check if a query is covered is to build binary expression objects out of expression tree and then check that all the simple expressions are present in the cached query and that the compound queries (`$and` and `$or`) are same or cached is more relaxed $(or)thannew(\text{and})$.

| BINARY EXPRESSIONS | | |
|---|---|---|
| **Simple** | | |
| 6:name | 5:$regex | 7:Beach |
| 9:property_type | 8:= | 10:House |
| 12:bedrooms | 11:= | 13:3 |
| 15:beds | 14:= | 15:5 |
| 18:guests_included | 17:= | 19:6 |
| **Compound** | | |
| 5:$regex | 3:$and | 8:= |
| 11:= | 4:$or | 14:= |
| 14:= | 4:$or | 17:= |

Figure 1: Binary Expressions

3. If a matching query is not found (cache miss), then we execute query on

the remote server (MongoDB cloud). The result of the query is stored in the result cache and the query itself is stored in the query cache (more details later).

4. On the other hand, if a matching query is found (cache hit), then we fetch all the result records from the result cache whose query id equals the matching query id. Next each result record is evaluated against the expression tree of the new query to check if it satisfies the expression. This is important so that in case of cached query not being identical to the new query, irrelevant result records are discarded. The logic for this resides in `MongoDBQueryCache.CachingMongoDbProxy::ResultSatisfiesCondition()`.

## Cache Design

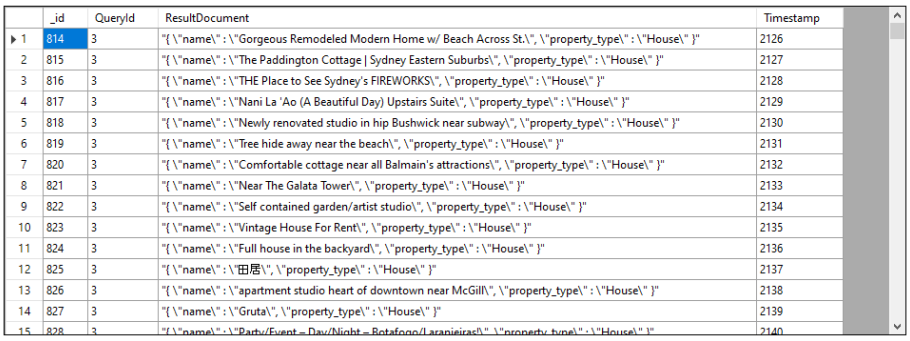
We use LiteDB, a serverless database delivered as a small library. LiteDB is a non-SQL database. We have two caches - `a query cache` and `a result cache`.

The `Query cache` table :

| | _id | QueryJsonString | Timestamp |
|---|---|---|---|
| ▶ 1 | 3 | "[\r\n {\r\n \"$match\": {\r\n \"$or\": [\r\n { \"name\": { \"$regex\": \"Beach\" } },\r\n { \"property_type\": \"House\" }\r\n ... | 7 |

Figure 2: Query Cache

The `Results cache` table :



| | _id | QueryId | ResultDocument | Timestamp |
|---|---|---|---|---|
| ▶ 1 | 814 | 3 | "{ \"name\" : \"Gorgeous Remodeled Modern Home w/ Beach Across St.\", \"property_type\" : \"House\" }" | 2126 |
| 2 | 815 | 3 | "{ \"name\" : \"The Paddington Cottage \| Sydney Eastern Suburbs\", \"property_type\" : \"House\" }" | 2127 |
| 3 | 816 | 3 | "{ \"name\" : \"THE Place to See Sydney's FIREWORKS\", \"property_type\" : \"House\" }" | 2128 |
| 4 | 817 | 3 | "{ \"name\" : \"Nani La 'Ao (A Beautiful Day) Upstairs Suite\", \"property_type\" : \"House\" }" | 2129 |
| 5 | 818 | 3 | "{ \"name\" : \"Newly renovated studio in hip Bushwick near subway\", \"property_type\" : \"House\" }" | 2130 |
| 6 | 819 | 3 | "{ \"name\" : \"Tree hide away near the beach\", \"property_type\" : \"House\" }" | 2131 |
| 7 | 820 | 3 | "{ \"name\" : \"Comfortable cottage near all Balmain's attractions\", \"property_type\" : \"House\" }" | 2132 |
| 8 | 821 | 3 | "{ \"name\" : \"Near The Galata Tower\", \"property_type\" : \"House\" }" | 2133 |
| 9 | 822 | 3 | "{ \"name\" : \"Self contained garden/artist studio\", \"property_type\" : \"House\" }" | 2134 |
| 10 | 823 | 3 | "{ \"name\" : \"Vintage House For Rent\", \"property_type\" : \"House\" }" | 2135 |
| 11 | 824 | 3 | "{ \"name\" : \"Full house in the backyard\", \"property_type\" : \"House\" }" | 2136 |
| 12 | 825 | 3 | "{ \"name\" : \"田居\", \"property_type\" : \"House\" }" | 2137 |
| 13 | 826 | 3 | "{ \"name\" : \"apartment studio heart of downtown near McGill\", \"property_type\" : \"House\" }" | 2138 |
| 14 | 827 | 3 | "{ \"name\" : \"Gruta\", \"property_type\" : \"House\" }" | 2139 |
| 15 | 828 | 3 | "{ \"name\" : \"Party/Event – Day/Night – Botafogo/Laranjeiras\", \"property_type\" : \"House\" }" | 2140 |

Figure 3: Results Cache

### LRU policy implementation

- We have implemented min-heap for LRU. The min-heap is keyed by timestamp. When an item as inserted in the cache, we put the timestamp of insertion.
- Whenever there is a cache hit and one or more cache items are re-used, we update the timestamp field of the cache entry with the access time.

- When time to evict an entry comes, exactract operation on min-heap efficiently tells us which item should be evicted (i.e. one with least time).
- When a query is evicted from `Query cache`, all the results which were fetched from remote server by that query are evicted from the `results_cache`. Result cache takes more space, so it's imperative to remove the results.

## Why we didn't exploit query rewriting to full extent

1. Suppose the original query, query1, was: `"$match": { "$or": [ { "name": { "$regex": "Beach" } }, { "property_type": "House" } ]}` This can be written as `A $or B`.
2. And the new query, query2, is: `"$match": { "$and": [ { "name": { "$regex": "Beach" } }, { "property_type": "House" } ]}` This can be written as `A $and B`.
3. One way to check if query2 is satisfied by query1 could be: `SELECT * FROM cache WHERE query_conditions CONTAINS('A $and B') OR query_conditions CONTAINS ('A $or B')`. This correctly gives the result that query1 satisfies query2.
4. But, the pitfall is that this query would also give a match if query1 were `A $and B $and C`, which would be incorrect.
5. For the above reason, we evaluate satisifiability in code by matching the individual simple expressions as well as walking up the tree to check it at every node.
6. Now coming to results, once we have a correct matching query, we can get the correct result set by running a query like this: `SELECT * FROM cache WHERE A AND B`, where A may be `name LIKE '%Beach%'` and B `property_type = 'House'`.
7. However, the above requires that each column of the record is stored as a separate column in a relational table. This is problematic if we want our caching solution to be general and not tied to any particular application schema (airbnb listings). Alternatively, the database should support extending the schema on-the-fly as new records with more columns than before are inserted.
8. The above could be solved by combining SQL query with power of some other query language, such as a JSON query language. For example, a hypothetical query could look like: `SELECT * FROM cache WHERE result_json SATISFIES JSON_QUERY ("$.name ~ 'Beach' and $.proerty_type = 'House')` where `result_json` is stored a single column of the table.

9. While this is an optimal idea, and LiteDB may as well be supporting something similar, due to learning curve involved, we choose to fetch the complete `result_json` in application and evaluate against the query conditions ourselves, as discussed in point 4 of Flow section above.