LiteIndex: Memory-Efficient Schema-Agnostic Indexing for JSON documents in SQLite

ASP-DAC 2021

Siqi Shang¹, Qihong Wu², Tianyu Wang³ and Zili Shao¹

¹ The Chinese University of Hong Kong, Hong Kong, China
I am currently a senior year undergraduate at the Chinese University of Hong Kong (CUHK). My main research interests include systems, robotics and Intelligence.
Outline

➢ Background and Motivation
➢ LiteIndex
   ➢ Memory-efficient Automatic Indexing
   ➢ Keywords Extraction from Long texts
   ➢ Pruning Policy
➢ Evaluation
➢ Conclusion
JSON (JavaScript Object Notation) is a lightweight data-interchange format structured as key/value pairs.

Most of the mobile applications take SQLite as their back-end storage engine for semi-structured JSON data.

A JSON document sample from Twitter:

```json
{"created_at" : "Thu May 10 15:24:15 +0000 2018" ,
"id_str" : "850006245121695744" ,
"text" : "Here is the Tweet message." ,
"user" : {},
"place" : {},
"entities" : {},
"extended_entities" : {} }
```
JSON in SQLite

- SQLite provides extension called "JSON1" to parse JSON data.
  - Cons: Keep full JSON record and parse for every query

- SQLite provides full text search extensions named FTS3 (Full Text Search) and FTS4 to create full-text.
  - Cons: Large memory consumption, difficult to extract the schema of JSON

- Developers can manually parse JSON documents and then create a schema for building a database.
  - Cons: Manually create indexes and can not handle keywords.
Schema-agnostic Indexing

- JSON documents as trees
- Azure DocumentDB: Triple-level terms
Motivation

- Challenges:
  - Indexing for semi-structured JSON records
  - Limited memory space on mobile devices
  - Long query latency

- We propose LiteIndex with three components:
  - Memory-efficient index organization
  - Lightweight keyword extraction from long text
  - User-preference-aware index pruning
Outline

➢ Background and Motivation

➢ LiteIndex
  ➢ Memory-efficient Automatic Indexing
  ➢ Keywords Extraction from Long texts
  ➢ Pruning Policy

➢ Evaluation

➢ Conclusion
LiteIndex System Architecture

New JSON document arrives

Indexing by LiteIndex

Store into SQLite

Add the new row (Document ID, Document Content) to the table

Schema Agnostic Indexing

Append the new name to the name set

Add the new value content to the value set

Insertion

Through Hashing

Name Table

LiteIndex Data Structure

Searching index

New query arrives

Keywords Extraction

Generate keywords’ indexes and insert into keyword table

Insertion

Through Hashing

Value Trees

Pruning Policy

Update the indexes when reaching the memory usage threshold

Insertion

Through Hashing

Keyword Table

Pruning

If not found, query database & perform full text search
Memory-efficient Automatic Indexing

Divide **names** and **values** in JSON

<table>
<thead>
<tr>
<th>Name set</th>
<th>Value set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only hundreds of attributes</td>
<td>Occupy most space</td>
</tr>
<tr>
<td>Stored in a hash table</td>
<td>Stored in smaller B+ trees</td>
</tr>
<tr>
<td>Serve as roots of value trees</td>
<td>Stores document IDs</td>
</tr>
</tbody>
</table>

- **Size** of the name set should be **much smaller**
- **More efficient search with less space consumption!**
Indexing Example

- **Name set** Value set
- **Query:**
  - Search in the name table to find corresponding value tree
  - Get the document id list from the value tree
- **Example:**
  
  ```sql
  SELECT id FROM jsonTable WHERE json_extract(file,'$.user.screen_name') = 'Adam';
  ```
Keywords Extraction from long texts

- **Not efficient** to directly hash the text into few bytes and use them as index
  - Basically *useless*

- Word embedding and clustering
  - Requires much **memory overhead** and **computing power**

- We propose a less computation-intensive keyword extraction method based on **TF-IDF** (term frequency–inverse document frequency)
  - **Threshold** selection
  - Take **more** keywords
Keywords Extraction from long texts

Given a JSON document text set $T$, a word $w$, and an individual text in JSON document $t \in T$, the TF-IDF value is calculated as follows:

$$w_t = f_{w,t} \times \log \left( \frac{|T|}{f_{w,T}} \right)$$

Here, $f_{w,t}$ is the frequency of $w$ appearing in $t$, $|T|$ is the size of the text set, $f_{w,T}$ is the number of documents containing $w$ in $T$.

Threshold Selection

- Semantic keywords in TF-IDF ≠ keywords in queries
- Memory overhead?
- Stored in keywords table with corresponding document id list.
Pruning Policy

- Apply approximate Q-learning technique to learn the user query preference

- **States**
  - Num of indexes in each value tree
  - Num of indexes in the keyword table

- **Q-value** function:
  \[ Q(s,a) = w_1 \cdot f_1(s,a) + w_2 \cdot f_2(s,a) + \ldots + w_n \cdot f_n(s,a) = \vec{w} \cdot \vec{f}(s,a) \]

  - \( w_n \) is the weight and \( f_n(s,a) \) is the number of indexes in value trees or keyword table. The learning begins with state \( s \) and then applies Action \( a \).
Pruning Policy

- The difference:
  \[ \text{difference} = [R(s, a, s') + \gamma \max_{a'} Q(s', a')] - Q(s, a) \]
  - \(s'\) and \(a'\) represents the next state and action of the state, \(R\) is the true reward of taking action \(a\) on state \(s\) to \(s'\), and \(\gamma\) is the discount factor.

- Update rule:
  \[ w_i \leftarrow w_i + \alpha \cdot \text{difference} \cdot f_i(s, a) \]
  - \(\alpha\): learning rate
Pruning Policy - actions

- Insertion Pruning:
  - **Delete** indexes from value trees or the keyword table and store them on disk
  - **Random** pruning for value trees
  - Keep scores for keywords
    - Considering both LRU (Least Recently Used) and LFU (Least Frequently Used):
      \[
      \text{score} = F\text{score} + R\text{score}
      \]

- **Replace**
  - **Recover** all the indexes in the value tree with highest weight
  - **Prune** indexes from the value tree with lowest weight
Outline

➢ Background and Motivation
➢ LiteIndex
  ➢ Memory-efficient Automatic Indexing
  ➢ Keywords Extraction from Long texts
  ➢ Pruning Policy
➢ Evaluation
➢ Conclusion
Experiment Setup

- Android Emulator with the setup of Nexus 6 phone
- 1536MB memory and 4 cores CPU
- 5GB twitter dataset from Jan 2019 to June 2020
- Around 200 name/value pairs with 5.4KB size per tweet
- Feed the tweets to mobile application
Overall Performance

- **LiteIndex** can significantly reduce the query latency by up to **18x** and the average latency reduction is **8.2x**

### Table 1: Query set description.

<table>
<thead>
<tr>
<th>Name</th>
<th>SQL Query Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>KWfT</td>
<td>SELECT DID FROM table WHERE text MATCH &quot;keyword&quot;</td>
</tr>
<tr>
<td>IntComp</td>
<td>SELECT DID FROM table WHERE favourites_count &gt; 50000</td>
</tr>
<tr>
<td>LPFR</td>
<td>SELECT DID FROM table WHERE json_extract(tweet,'$.user.name')='Bob'</td>
</tr>
<tr>
<td>PS</td>
<td>SELECT DID FROM table WHERE location = 'Paris'</td>
</tr>
</tbody>
</table>

### Table 2: Comparison of query latencies (ms) between SQLite-FTS (SQLite with FTS3/FTS4 extensions) and LiteIndex.

<table>
<thead>
<tr>
<th>Query Set</th>
<th>SQLite-FTS</th>
<th>LiteIndex</th>
<th>Latency Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>KWfT_s</td>
<td>213</td>
<td>79</td>
<td>2.7x</td>
</tr>
<tr>
<td>IntCmp_s</td>
<td>19</td>
<td>2</td>
<td>9.5x</td>
</tr>
<tr>
<td>LPfR_s</td>
<td>18</td>
<td>2</td>
<td>9x</td>
</tr>
<tr>
<td>PS_s</td>
<td>40</td>
<td>5</td>
<td>5x</td>
</tr>
<tr>
<td>KWfT_l</td>
<td>207</td>
<td>113</td>
<td>1.8x</td>
</tr>
<tr>
<td>IntCmp_l</td>
<td>19</td>
<td>2</td>
<td>9.5x</td>
</tr>
<tr>
<td>LPfR_l</td>
<td>20</td>
<td>2</td>
<td>10x</td>
</tr>
<tr>
<td>PS_l</td>
<td>54</td>
<td>3</td>
<td>18x</td>
</tr>
</tbody>
</table>

- **LiteIndex** can significantly reduce the query latency by up to **18x** and the average latency reduction is **8.2x**
LiteIndex achieves up to 4x index size reduction compared with Azure DocumentDB.
Keyword Selection

- Proportion of 0.5 provides 0.83 recall value, which is close to Brown Clustering WordVec method.
Pruning Policy

Our pruning policy can efficiently learn about user’s new preferences.
Conclusion

❑ Previous Methods:
  ❑ Manual table setup, more space usage and high query latency

❑ We propose LiteIndex:
  ❑ A novel memory-efficient automatic indexing scheme
  ❑ Lightweight keyword extraction from long texts
  ❑ User-preference-aware index pruning policy

❑ We achieve:
  ❑ Reduce the query latency by up to 18x
  ❑ Restrict the memory usage under 5MB