Taxonomy Of Online Update Techniques For Column Store Database

Presented by: Udit Panchal & Dhruv Patel

CS848 : Main/In-Memory Database Systems | Spring 2016
University of Waterloo
Overview

- Columnar Database and Challenges
- Motivation
- Overheads
- Contribution: Taxonomy of Online Updates
- Data Structures
- Examples
Columnar Database and Challenges

- Why we need Columnar Database System?
  - Analytical query sessions inspect large amount of queries, but on small subset of columns.
  - Thus, OLAP are not suitable for row-store architecture.
  - Columnar storage avoids data access for unused columns, improving cost of the query.

- Following approaches further increase performance:
  - Various projection of same data, in various orders
  - Data Compression, reducing data access cost
Columnar Database and Challenges

- But read-optimized databases are not write friendly
- Many disk I/O required for a single write query.
Columnar Database and Challenges

- But read-optimized databases are not write friendly
  - Many disk I/O required for a single write query.
  - Compression makes it more computationally expensive & complex
    - Data needs to be de-compressed, updated and recompressed
    - Extra complication occurs if the updated data no longer fits its original location
  - Maintain reasonable physical interference between concurrent queries and updates.
Motivation

- **Scenario 1: Location based Mobile Advertising**
  - $18 billion annually
  - Location, shopping pattern, past purchase, browsing history
  - Aim: Improve effectiveness of future advertisements
    - Recent purchase must be available immediately to subsequent analytics

- **Scenario 2: Credit Card Fraud Detection**
  - $400 billion annually
  - Approve transaction in short span
    - Detect if transaction is fraudulent or not
    - Run complex analytics in real time
  - Match application with appropriate database management system
    - Difficult with increasing complex business
    - Up to date transactional data
Overheads

- Function of Latency, Throughput and Data Freshness.
  - Maintaining the indices for the delta, stored in the RAM
  - Merging of Delta to Main
    - Decompress and recompress the data in disk to merge update
    - Space consumption of the RAM to store delta

- When is the optimal time to merge data?
  - Size of the memory.
  - Size of the delta stored.
  - Physical interference between Delta and Main
  - Approach used i.e. with merge is blocking or not blocking the analytical queries.
Contribution: Taxonomy of Online Updates

- Architectural
  - In-Place
  - Differential

- Data Structure
  - SSD
  - HDD
  - RAM
  - Cache
  - Delta Layout
  - Row Based
  - Column Based

- Storage Place
  - SSD
  - HDD
  - RAM
  - Cache

- Merge Approach
  - Explicit Merge
  - Implicit Merge
  - Merge during scan

- Merge during scan

- Merge Approach

- Architectural
  - In-Place
  - Differential

- Data Structure
  - SSD
  - HDD
  - RAM
  - Cache
  - Delta Layout
  - Row Based
  - Column Based

- Storage Place
  - SSD
  - HDD
  - RAM
  - Cache
Data Structures

- Positional Delta Tree
- Value based Delta Tree
- Count Index
- Log Structured Merge Tree
- Differential Files
- and more data structures covered in report
Differential Files

- File consists two parts
  - Main File: unchanged
  - Differential File (DF): records all alterations requested for the main file
    - PID, time-stamp, other identification information
- Retrieve: DF always searched first
  - May not be present in DF
  - BitMap accessed by hashing scheme: reduce probability of making unnecessary search
    - If bits are set to 1: may be record in DF otherwise skip and go to Main File
    - If bits are set to 1 coincidentally by mapping from other updated records, search will be fruitless
- DF must be merged with main file
  - Based on threshold
  - DF will be empty again
Example: Differential Files

VADIS System
- Architecture: Differential
- Data Structure: Differential Files
- Storage Place: RAM & HDD
- Merge Approach: Explicit - Bulk merge when reached to threshold (I/O intensive)
- Delta Layout: Row based
Log Structured Merge Tree

- Disk based data structure
  - Provides low cost indexing for files experiencing high rate of record insert
  - Reduce disk arm movement
    - Sequentially organized logical data
  - Memory resident C0, Disk resident C1
- Delete:
  - If key-value entry is not found in C0
    - Place *delete node entry*, indexed by key value
    - Delete will be performed at rolling merge process
  - What if find request comes in meantime?
    - Delete filter
    - Delete node entry will be located in the appropriate key-value position than the entry itself
Log Structured Merge Tree

- **Update:**
  - Updates are considered as delete followed by insert
  - Change the index value - mostly unusual in other applications
- **Predicate Deletion:**
  - Perform batch delete
  - Assert a predicate
  - Performed during rolling merge
- **Long latency Find:**
  - *find note entry* inserted into C0
  - Find is performed at later time during merge process
- **Insert:**
  - Inserting an index entry has no I/O cost
- **Merge process happens on threshold value:** series of merge steps
Example: LSM Tree

Jin et. al. developed plug-in system PuntStore with pLSM
- Variant of LSM: pLSM
- Uses COLA (Cache Oblivious Lookahead Array)
- Architecture: Differential
- Data Structure: LSM
- Storage Place: RAM & HDD
- Merge Approach: Merge with scan
- Delta Layout: Row based
Value-based Delta Tree (VDT)

- VDT: B+-Tree sorted on the key values.
- In MonetDB, we have insert table & delete table which stores all the relevant records, in append-only fashion.
- Example: When we apply filter on X Column, there will be one select operator applied directly on column X and another select operator applied on the pending updates columns of X, while subsequently qualifying pending inserts are merged or pending deletes are removed from the corresponding intermediate result.

<table>
<thead>
<tr>
<th>store</th>
<th>prod</th>
<th>new</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berlin</td>
<td>chair</td>
<td>Y</td>
<td>20</td>
</tr>
<tr>
<td>Berlin</td>
<td>cloth</td>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>Berlin</td>
<td>rack</td>
<td>Y</td>
<td>4</td>
</tr>
<tr>
<td>London</td>
<td>rack</td>
<td>Y</td>
<td>4</td>
</tr>
<tr>
<td>London</td>
<td>stool</td>
<td>N</td>
<td>9</td>
</tr>
<tr>
<td>Paris</td>
<td>rug</td>
<td></td>
<td></td>
</tr>
<tr>
<td>London</td>
<td>stool</td>
<td>Y</td>
<td>4</td>
</tr>
</tbody>
</table>

(a) Insertion table

<table>
<thead>
<tr>
<th>store</th>
<th>prod</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paris</td>
<td>rug</td>
</tr>
<tr>
<td>London</td>
<td>stool</td>
</tr>
</tbody>
</table>

(b) Deletion table
Value-based Delta Tree (VDT)

- But,
  - When the read queries are executed, they need to merge the differences by looking at the sort key values.
    - That is, first they have to look for the sort key values, which can be time consuming and requires more I/O.
    - Thus, affects throughput and performance of the DBMS
Positional Delta Tree (PDT)

- PDT: PDTs are similar to counted B-Trees,
  - Provides fast merging of the updates, by providing tuple positions, where differences have to be applied at update time.
  - The leaf node of the PDT stores the SID where the updates applies, type of update and reference to the new tuple values.
  - Since the values of each update type is different, each of those are stored in separate “value tables”
Positional Delta Tree (PDT)

Figure 1: TABLE_0

Figure 2: BATCH_1

Figure 3: PDT_1

Figure 4: VALS_1
Positional Delta Tree (PDT)

Figure 5: TABLE

Figure 6: BATCH

Figure 7: PDT

Figure 8: VALS
Positional Delta Tree (PDT)

- Advantages of PDT over VDT
  - No need to read sort keys during merging, thus reducing I/O operations
  - Positional merging is less CPU intensive, than VDT based merging.
Count Index

- Use of PDT and VDT makes a database two-pronged store: a read-optimized (rs) and a write-optimized (ws) store.
  - Buffers update in the differential store and later merges the updates with rs
  - Bulk update amortizes the time to apply, but cannot avoid the linear cost of table scan
  - While merging, we have to decompress and subsequently re-compressed the data which requires additional time
- Can’t we avoid buffering of the updates and directly update the data in-place?
  - Count Index does so, in sub-linear time complexity of total number of tuples.
Count Index

- So in nutshell
  - Count Index are in-memory index for run-length encoded in-memory/disk database system.
    - Count index is a binary tree on a sequence of integers, where integer is the sum of the values of its children.

![Count Index on the sequence a, a, b, b, a, a, b, b](image)

*Figure 1: Count index on the sequence* a, a, b, b, a, a, b, b
Count Index

- So in nutshell
  - Count Index are in-memory index for run-length encoded in-memory/disk database system.
    - Count index is a binary tree on a sequence of integers, where integer is the sum of the values of its children.
  - Any index, should be efficiently updatable, when values are updated, or deleted or new values inserted
    - Deleting or inserting a leaf from a count index takes time that is linear in the height of the count index
Advantages of Count Index over PDT

- **Space complexity**
  - If $n$ is total number of tuples in relation and we add $k$ tuples, after which we delete those $k$ tuples, then the space required to store the corresponding PDT would be $O(n + k)$, whereas for Count Index would be $O(n)$.

- **Merge time complexity**
  - PDT requires a merge scan to bulk insert values and thus time complexity of updating a relation is $O(n)$.
  - Whereas count index requires $O(k \times \log n)$.

- Using PDT we need to de-compress and later re-compress the column data, while in count index, we can operate directly on the compressed sequences of values.

- **Maintaining the PDT is computationally expensive, as**
  - PDT stores number of position-based meta-data which needs to be changed, during various update types.
Examples

Analytics in Motion

- Architecture: Differential
- Data Structure: Delta in Google’s Dense Hash Map & Main in ColumnMap (Key Store)
- Storage Place: RAM
- Merge Approach: Explicitly triggered too often to avoid blocking of RTA
- Delta Layout: Hash Map
Examples

Hyper

- Architecture: In-Place Updates
- Data Structure: Snapshot of virtual memory
- Storage Place: RAM
- Merge Approach: Copy on demand
- Delta Layout: Row-based
Examples

Hyper [RAM conservative]

- **Architecture**: In-Place Updates
- **Data Structure**: Snapshot of virtual memory
- **Storage Place**: RAM
- **Merge Approach**: Copy on demand
- **Delta Layout**: Compressed and Column-based
Examples

SQL Server - 2016

- Architecture: Differential Update
- Data Structure: VDT for update and insert & bitmap for delete
- Storage Place: RAM
- Merge Approach: Explicitly triggered because compressed data has efficient space usage and query execution
- Delta Layout: Row Store.
Conclusion

- Searched and categorized existing work on online updates
- Ongoing: compare and contrast approaches and data structures