Wave-Indices: Indexing Evolving Databases

Hector Garcia-Molina – Stanford University
Narayanan Shivakumar – Stanford University

Presentation By: Mirza Beg

Outline
- Problem Statement
- Formalisms
- Proposed Techniques
- Analytic Comparison
- Case Studies
- Conclusion
- Discussion

Problem Statement
- Large amounts of data being generated everyday
- Need to index into the data of some past window of days:
  - Search engines – provide an index for the past 30 days of some news articles.
  - Financial institutions – keep an index of the stock market trades for the past 7 days
- Each day, a new batch of data must be added to the index, and data older than the window should be removed.
- Where would we need this?
  - Application semantics require sliding window – Streams
  - Users interest wanes over time – News
  - Want to reduce storage costs – Too much data -> slow access
Sliding Window Indices

- Have been in use for many years
- Need to revisit existing schemes because of the increasing volumes of data
  - Web Search: Engines need to keep track of ever-increasing web pages, articles, and other information.
  - Data Warehousing: huge volumes of sales, banking, and other transactions need efficient access to information.
  - SCAM (Stanford Copy Analysis Mechanism): detecting illegal copies of copyrighted digital documents.

Solution:
- Wave Indices

What are Wave Indices?

Traditional Indexing:
- Keep a single conventional index, and every day delete the old batch of data and insert the new batch of data into it.
- \( I = \{ d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9, d_{10} \} \)
- Operation update: add \( d_{11} \) and delete \( d_1 \)
- Wave Index:
  - Index a window of \( W \) days and partition the data across multiple indices.
  - Create \( W/n \) indices where \( n \) is the number of constituent indices.
  - Service queries by accessing all indices.
  - Update operations and techniques.

Index Structures

- Data we need to index exist as records \( r_1, r_2, \ldots \)
- Each of the records has a search field \( F \)
- Index is built on the search field
- Each record may have multiple values for \( F \)
- Thus multiple buckets may be referencing the same record.
- Index consists of a directory and associated buckets.
  - Directory is a search structure.
  - Given a value \( v \) identifies a bucket \( b \)
  - Each bucket has a pointer \( p \) to record \( r_i \)
  - and may have additional information \( a_i \)
  - Queries can possibly scan the whole index (aggregate functions).

*Directory is in memory and buckets are stored contiguously on disk.*
Index Update Techniques

- **In-Place Updating**
  - The directory and/or buckets are modified in place.
  - If bucket is full, then bucket is copied to a new location and allocated more space.
  - Resulting index is not packed even if the original was packed.
  - Concurrency control required during updates.

- **Simple Shadow Updating**
  - Makes a copy of the index
  - For each update modifies the new copy of the index in place
  - The new index replaces the old version in the wave index
  - No concurrency control required.

- **Packed Shadow Updating**
  - Same as Simple Shadow except that the resulting index is packed.

*packed – an index is said to be packed if each of its buckets are allocated contiguously on disk and use the internal amount of space to store entries.

Operations in Building Wave Indices

- **Primitive Functions on a Wave Index θ**
  - **AddIndex(Ι,i)** – Adds Ι to the set of constituent indices in θ.
  - **DropIndex(Ι,i)** – Removes Ι from θ. Deletes all entries in Ι.
  - **BuildIndex(Days,Ι)** – Builds a packed index Ι for the batch of records in those days.
  - **AddEntries(Days,Ι)** – Incrementally adds the batch of entries for Days records to Ι.
  - **DeleteFromIndex(Days,Ι)** – Incrementally deletes entries for Days from Ι.
  - **TimedIndexProbe(θ,T₁,T₂,s)** – For a wave index θ, T₁ and T₂ retrieves buckets of entries for s inserted between T₁ and T₂.
  - **TimedSegmentScan(θ,T₁,T₂)** – Retrieves all entries inserted between T₁ and T₂.

DEL - Deletion based

- Initially index θ with days of data in indices Ι₁, ..., Ιₙ.
- Then make Ι₁, ..., Ιₙ constituent indices of wave index θ.
- When dᵢᵢ is available, delete entries of dᵢᵢ from Ιᵢ that indexed dᵢᵢ in θ.
- Then we insert entries for dᵢᵢ to Ιᵢ.
- DEL maintains hard windows.

<table>
<thead>
<tr>
<th>Day</th>
<th>Old Index</th>
<th>New Index</th>
<th>Time</th>
<th>Weekly Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ι₁</td>
<td>Ι₂</td>
<td>1</td>
<td>Ι₁</td>
</tr>
<tr>
<td>2</td>
<td>Ι₂</td>
<td>Ι₃</td>
<td>1</td>
<td>Ι₂</td>
</tr>
<tr>
<td>3</td>
<td>Ι₃</td>
<td>Ι₄</td>
<td>1</td>
<td>Ι₃</td>
</tr>
<tr>
<td>4</td>
<td>Ι₄</td>
<td>Ι₅</td>
<td>1</td>
<td>Ι₄</td>
</tr>
</tbody>
</table>

Table 1: Deletion based index management (θ’ = ι₁’).
REINDEX - Reindexing

- Initially index W/n days of data each in indices Η₁, ..., Ηₙ.
- Then make Η₁, ..., Ηₙ constituent indexes of wave index θ.
- When dₙew is available, delete all entries Ηᵢ that indexed dₙew⁻¹.
- Then we rebuild Ηᵢ with entries for dₙew⁻¹, ..., dₙew.

<table>
<thead>
<tr>
<th>Time Index</th>
<th>Temporary</th>
<th>Data</th>
<th>Time Index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Reindexing based index maintenance (W = 1, n = 2).

- When data dᵢ₁ is available replace dᵢ in Ηᵢ by dᵢ₁ by rebuilding index Ηᵢ with data dᵢ, dᵢ₂, dᵢ₃, and dᵢ₄. Similarly with subsequent days.
- Maintains hard windows.
- Requires reindexing W/n days of data every day.

REINDEX⁺ - REINDEX

Enhanced

- REINDEX⁺ maintains a temporary index, Temp.
- Avoids re-computing index entries everyday.

<table>
<thead>
<tr>
<th>Time Index</th>
<th>Temporary</th>
<th>Data</th>
<th>Time Index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Example of Index Transition in REINDEX⁺ (W = 1, n = 2).

- On average reindexes about half the number of days REINDEX does.

REINDEX++ - Enhanced REINDEX⁺

- Maintains more than one temporary index {T₁, ..., Tₙ}
- Performs most of the maintenance work for the wave index in advance.

<table>
<thead>
<tr>
<th>Time Index</th>
<th>Temporary</th>
<th>Data</th>
<th>Time Index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Example of Index Transition in REINDEX++ (W = 1, n = 2).

- Increased storage requirements at the cost of making data available sooner.
- Does about the same amount of work as REINDEX⁺ but reduces time to index a new day's data.
**WATA - Wait And Throw Away**
- Uses lazy form of deletion by throwing away an entire index when all its entries have expired.
- When new data is available, add it to index with unused capacity.
- When no such index available, first throw away index with the oldest data.

<table>
<thead>
<tr>
<th>Day</th>
<th>New Data</th>
<th>Operation</th>
<th>Retain</th>
<th>Expire</th>
<th>New Retain</th>
<th>New Expire</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>d1</td>
<td>Reindex</td>
<td>Ι1</td>
<td>Ι4</td>
<td>Ι2</td>
<td>Ι3</td>
</tr>
<tr>
<td>2</td>
<td>d2</td>
<td>Reindex</td>
<td>Ι2</td>
<td>Ι4</td>
<td>Ι1</td>
<td>Ι3</td>
</tr>
<tr>
<td>3</td>
<td>d3</td>
<td>Reindex</td>
<td>Ι3</td>
<td>Ι4</td>
<td>Ι1</td>
<td>Ι2</td>
</tr>
<tr>
<td>4</td>
<td>d4</td>
<td>Reindex</td>
<td>Ι4</td>
<td>Ι1</td>
<td>Ι2</td>
<td>Ι3</td>
</tr>
</tbody>
</table>

- For the first 10 days index data into Ι1, ..., Ι4.
- When data d1 is available, add it to Ι1. Similarly for d2. When data d3 is available on the 12th day, first throw away Ι1, then create a new index Ι1, and finally add d3 to it. The next day add d4 to Ι1, and so on.
- Maintains soft windows -> Uses more space -> Relatively little work each day.
- Requires at least two constituent indices.

**RATA - Reindex And Throw Away**
- A hybrid of REINDEX++ and WATA.
- Uses temporary indices S to be computed in advance to replace some Ιj.

<table>
<thead>
<tr>
<th>Day</th>
<th>New Data</th>
<th>Operation</th>
<th>Retain</th>
<th>Expire</th>
<th>New Retain</th>
<th>New Expire</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>d1</td>
<td>Reindex</td>
<td>Ι1</td>
<td>Ι4</td>
<td>Ι2</td>
<td>Ι3</td>
</tr>
<tr>
<td>2</td>
<td>d2</td>
<td>Reindex</td>
<td>Ι2</td>
<td>Ι4</td>
<td>Ι1</td>
<td>Ι3</td>
</tr>
<tr>
<td>3</td>
<td>d3</td>
<td>Reindex</td>
<td>Ι3</td>
<td>Ι4</td>
<td>Ι1</td>
<td>Ι2</td>
</tr>
<tr>
<td>4</td>
<td>d4</td>
<td>Reindex</td>
<td>Ι4</td>
<td>Ι1</td>
<td>Ι2</td>
<td>Ι3</td>
</tr>
</tbody>
</table>

- For the first 10 days index data into Ι1, ..., Ι4. Builds temp indices Τ0, Τ1.
- Indexes {d3} -> Τ0 and {d3, d2} -> Τ1. Later drops Ι1 and replace Ι1 with Τ0.
- Performs more work than WATA. Maintains hard windows.
- Takes the same time as WATA to make the data available.

**Analytic Comparison**
- Working with a window of W days.
- S is the space required to store a packed index of one day.
- S' is the space required to store a non-packed index of one day.
- REINDEX++ requires an average of \(W \times S'\) and a maximum of \((W+X-1) \times S'\) when it indexes constituent indices for W days. REINDEX++ Temp indexes at most \(X-1\) days. However when averaged is \(X/2\) days.
Case Studies

Illustration of performance trends and of the process for a particular wave index scheme

- **SCAM**
  - A research prototype for finding copyright violations.
  - For the experiments provides index to a set of newsgroups to allow authors to search for recent illegal copies of their articles.

- **Web Search Engine (WSE)**
  - Several WSE’s index a large set of WebPages for a sliding window of n previous days.
  - For a ‘generic’ WSE, report results for the case where WSE has to index articles for a sliding window of 35 days.

- **TPC-D**
  - A benchmark from the Transaction Processing council.
  - A large database modeling a decision support environment.
  - For SUPPKEY (att) on LINEITEM (rel), build a wave index for a window of the past 100 days. A single query is executed (‘Pricing Summary Report’).

Experimental Results 1

- Reports the overall space required (averaged across transitions).
- **REINDEX** requires the minimal space.
- Maintains packed indices.
- Does not have any temporary indices.
- Overall the schemes require less space as n increases.
- Fewer temporary indices required

- Transition time to index new data.
  - Using BuildIndex or AddIndex?
  - How many times are reindexed?
  - DEL, WATA, RATA, REINDEX execute AddIndex & Incrementally index 1 day.
  - REINDEX cost savings of BuildIndex if:
  - **REINDEX** performs poorly -> executes AddIndex several times each day

Experimental Results 2

- Total work done by different schemes.
  - Sensitive to the mix of queries and updates.
- Best to perform more work at update time in order to get better indexing packed.
- **REINDEX** performs well for large n
- Relative cost of BuildIndex + packed
  - DEL, WATA, RATA stable -> incremental additions

- Total work done by WSE with packed shadowing for W = 35.
  - **REINDEX** performs worst
  - Higher query volume and window size
  - DEL, WATA and RATA performs best and does the minimal total work – minimal work for reindexing new data
  - Index probes are cheap for small n.
Experimental Results 3

- Total work done in TPC-D with packet shadowing.
- REINDEX performs very poorly.
- DEL and WATA perform best.
  - Packet shadowing deletes immediately upon copying, reducing work.

- Total work done in TPC-D with simple shadowing.
- Performs very similar to packet shadowing, but REDUCE performs much better as \( n \) increases.
  - Number of expired days stored in the indices decrease as \( n \) increases.

Experimental Results 4

- How do schemes scale with increasing \( W \)?
  - DEL, WATA, RATA scale very well.
  - Index a constant number of days every day.
  - Since reindexing schemes index \( W/n \) days each day, work done increases with the size of \( W \).

- How does turnover rate affect the work done?
  - REINDEX scales best.
  - Does not use incremental indexing.
  - WATA still scales best for \( SF < 3 \).

Advantages of Using the schemes

- Bulk Insert/Delete: In WATA deletions are performed in bulk by throwing away a whole index. Similarly, it may be efficient to reindex large data, like in REINDEX, RINDEX+, and RINDEX++, if the constituent index size is reasonably small.

- Better Structured Index: REINDEX may be more costly because it rebuilds indexes from scratch, but this rebuilding can often lead to a better structured index. Such an index could lead to more efficient query processing.

- Simple Code: With REINDEX, RINDEX+, RINDEX++, WATA, and RATA, the schemes do not use complex deletion code. This could be a great advantage if we are implementing our system from scratch.

- Legacy Systems: Some information retrieval indexing packages do not implement deletes at all. In such cases, DEL cannot be used.

- Query Performance: Each scheme presented has multiple indexes, which creates more work for queries. However, when query volumes are relatively low, data volumes may be high, and each index is used by the average under some of the categories listed above.
Conclusions

- One of the first schemes to index data from a past window of days.
- Several techniques proposed for building indices on temporal data.
- Different techniques perform differently in different environments (volume of input data, query patterns, index lengths).
- In future, analyze how different indices perform when multiple disks are available.

Discussion Points

- Better techniques to index temporal data? Or maybe faster operations (e.g., Replace(\(t, d_{new}, d_{old}\))?)
- How would these schemes scale for frequently updated windows where tuple size is very small?
- Can they be used/modified to take care of bursty or out-of-order data (data from sensors on a highway)?
- Do the experiments appropriately demonstrate the performance of the different schemes?
- Is the scope too broad for one paper?

Questions?