Building and Configuring a Real-Time Indexing System

Garret Swart, Ravi Palakodety, Mohammad Faisal, Wesley Lin

SIGIR Geneva 2010

What does “Real-Time Indexing” mean?

- Transactional Indexing
  - When the document insert commits:
    - The document is durable and visible
    - The document has been indexed
    - Subsequent search requests will “see” the document
  - Required if the text index forms a required access path to the document
    - Is this important? Not usually. Required access paths are generally based on document meta data, not document contents

What does “Real-Time Indexing” mean?

- Timely Indexing of
  - Real time twits
    - @worldcup Gooooaaaaallll
  - Breaking news feeds
    - The Blick offers 20,000 CHF to find fugitive captain
  - Financial press releases
    - UBS AG reports first-quarter profit of 2.2 billion CHF
  - Email
    - **From:** Mom **Subject:** Meet Aunt Sophie at the airport
  - Shopping bargains
    - Today only! 25% off on Big Screen TVs
  - Facebook and LinkedIn status changes
    - Work those rebounds! Save those jobs!

What does “Real-Time Indexing” mean?

- Maintaining Search Quality
  - Deep document analysis
  - Global (Link and Anchor Text) analysis
  - Document rank prediction and validation
- Not subject of this talk
  - We’ll focus just on KWIC

In this talk

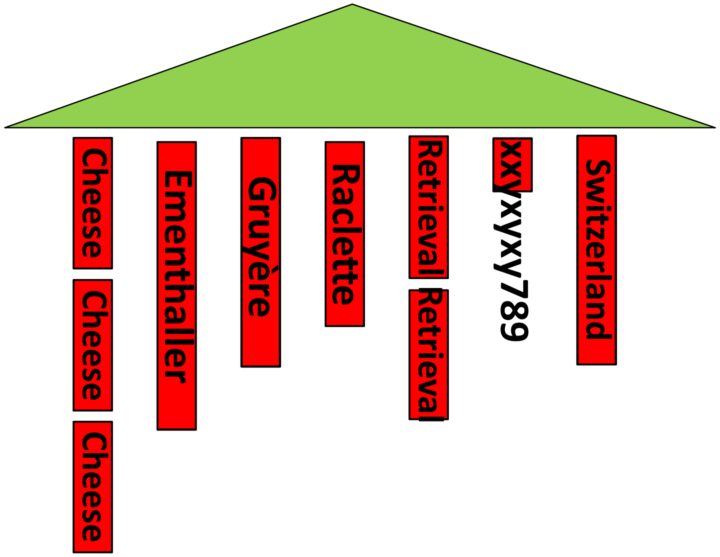
- Oracle Text background
- Real-time index maintenance in Oracle Text
- Performance modeling

Oracle Text

- A text processing, indexing and query facility for textual documents stored in an Oracle database
  - Originally released in 1997 as a cartridge for Oracle 8.0
- Documents are assigned increasing IDs as they are found. Documents can be:
  - CLOB and VARCHAR columns
  - Extracted from XML documents
  - Extracted from Word/PDF/JPEG files
  - Constructed with SQL or XQuery view
- Each document is parsed into index entries using a customizable parsing method
  - Each entry records a search term, the document ID, and the logical offset of the term inside the document

Oracle Text

- Let there be Indices
  - Inverted Lists: For each term we sort and compress the IDs and offsets and store them into a sequence of BLOBs, each stored contiguously
  - Terms and BLOB handles are stored in a B-Tree
- And the queries were good
  - SQL CONTAINS operator computes a relevance score based on a match of the pattern against the document
  - CONTAINS is implemented through parallel inverted list merging



```
SELECT * FROM (SELECT
title FROM News WHERE
CONTAINS(contents,
'NEAR((cheese,
Gruyère)) AND
Switzerland', 1) > 0
ORDER BY SCORE(1)
DESCENDING) WHERE
ROWNUM <= 10
```

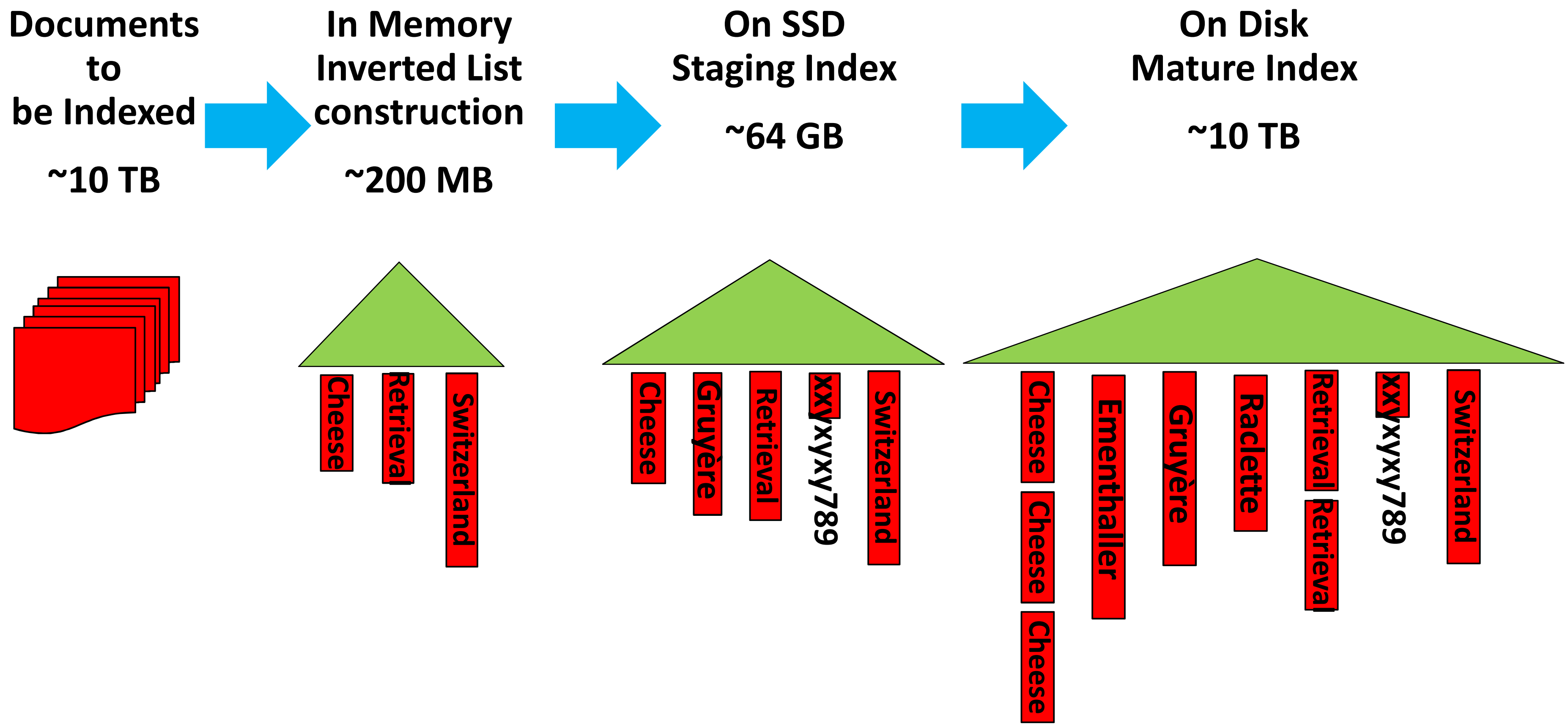
Text Index Update

- Text Indexes are updated in batches
  - As Index entries are produced they are grouped by term to form a set of in-memory inverted lists
  - When a transaction commits, memory fills up, or enough time has passed, each list is either
    - Appended to the end of an old BLOB
    - Written into a new BLOB and a new B-Tree entry created
- As the time between batches decreases
  - The number of entries per term decreases proportionally
  - The number of terms decreases slowly
  - The number of IOs increase
- In memory or SSD-only is not cost effective for low query/corpus-size applications

Real-Time Text Indexing

- Introduce a stable staging index for new index entries
- Logically merge the staging index and the mature index when searching
- In background, append staging BLOBs to the corresponding mature BLOB and delete the staging BLOB
  - Increase bg priority as the staging index gets larger
- Store the staging index on SSD and optionally pin its blocks in-memory
  - Storing staging index in memory avoids having to read the staging index when coalescing or querying

New Indexing Process



Documents to be Indexed

~10 TB

In Memory Inverted List construction

~200 MB

On SSD Staging Index

~64 GB

On Disk Mature Index

~10 TB

Evaluating the New Approach

- Is this good? It does more IO
  - Every index entry is written twice: Once to the staging and once to the mature index. But …
    - Staging updates are cheaper than mature updates:
      - Random updates to an SSD are cheaper
      - Updates to a smaller B-Tree are cheaper
    - Each write to the mature index adds many accumulated entries
- But unique terms don’t accumulate entries
  - Many documents have unique terms we must index
  - Solution: Hash unique terms to a set of “bucket terms”
  - Pull unique terms out of the bucket on repeated use
  - Rehash terms if an inverted list gets too long

Modeling Performance

- Why model performance? **Evaluate, Size, Tune**
  - **Evaluate** different algorithms
  - **Size**
    - DRAM, Storage system, CPUs
  - **Tune**
    - When should we add a new BLOB vs. append to an existing one?
    - How long should we let a bucket term’s inverted list grow?
- Managing performance?
  - DBA intervention
  - Performance “Advisors”
  - Self Management

Modeling Parameters

- Corpus Size
- Average Document Size
- Expected Distinct Terms: Size → Count
  - Derivative approaches a small non-negative constant
- Document Arrival Rate
  - Documents / Second
- Index Timeliness
- Terms and entries per term
  - Worst case (98%)
  - Expected case
- Query Performance
  - Worst case latency
  - Expected Throughput
- Maximum LOB size
- DRAM size & cost
- SSD size, IOPS, count, cost
- HDD size, IOPS, count, cost
- Document Parse CPU/MB
- Inverted List Merge CPU/MB
- Core Count

Constrain relationships between the parameters

- Worst case (98%) query latency
  - Many terms each with big uncached inverted lists
  - IO Latency:
    - BLOBs = Σ_{term} ceiling(|invertedList(term)| / BLOB-SIZE)
    - EMaxBallsPerBin(BLOBs, HDD count) * HDD Latency
- Expected query throughput
  - Fewer terms, many in cache
  - HDD throughput
    - BLOBs * MissRate / (HDD IOPS * HDD Count)
  - CPU throughput
    - *CPU/MB* * Σ_{term} |invertedList(term)| / Cores

Pick your inputs, chose your objective function, find a solution, validate

- Configure a given hardware configuration to give best throughput
- How to best utilize an extra $100K to improve worst case latency
- Size a new system to meet a new work load
  - Hard because parameters are hard to estimate for a new workload
- Caveats:
  - LP not good enough: Need nonlinear, integer programming
  - Easy to miss critical constraints, Need to validate answers

Conclusions/Assertions

- Real time indexing is useful for many applications and we want data indexed as fast as possible
- SSDs can change the set of suitable algorithms real-time indexing
- Performance modeling is good
  - Evaluate, Size, Tune
- Databases can support IR!
  - Database infrastructure is useful when building IR tools:
    - B-Trees, Write ahead logging, LOBs, Parallelism, Transactions, Disaster recovery, Encryption, Security, Extensibility
  - IR is essential in modern enterprise applications
  - IR applications can be built on an embedded database