CompSci 590.6

Understanding Data:

Theory and Applications

Lecture 4

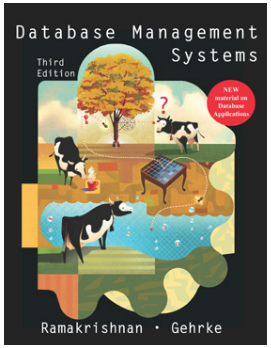
Data Warehousing
and
Iceberg Queries

Instructor: Sudeepa Roy
Email: *sudeepa@cs.duke.edu*

Today's Paper(s)

Chaudhuri-Dayal
An Overview of Data Warehousing and OLAP Technology
SIGMOD Record 1997

Book: Database Management Systems
Ramakrishnan-Gehrke
Chapter#25
Data Warehousing and Decision Support



Fang-Shivakumar- Garcia-Molina -Motwani-Ullman
Computing Iceberg Queries Efficiently
VLDB 1998

Data Warehousing (DW)

- A collection of decision support technologies
- To enable people in industry/organizations to make better decisions
  - Supports OLAP (On-Line Analytical Processing)
- Applications in
  - Manufacturing
  - Retail
  - Finance
  - Transportation
  - Healthcare
  - …

- Typically maintained separately from "Operational Databases"
  - Operational Databases support OLTP (On-Line Transaction Processing)

OLTP	Data Warehousing/OLAP
Applications: Order entry, sales update, banking transactions	Applications: Decision support in industry/organization
Detailed, up-to-date data	Summarized, historical data (from multiple operational db, grows over time)
Structured, repetitive, short tasks	Query intensive, ad hoc, complex queries
Each transaction reads/updates only a few tuples (tens of)	Each query can accesses many records, and perform many joins, scans, aggregates
Important: Consistency, recoverability, Maximizing transaction throughput	Important: Query throughput Response times

Terminology

- Multidimensional Data
  - Some dimensions are hierarchical (day-month-year)
- Operations
  - Roll-ups, Drill-down
  - Pivot (re-orient view) – attr value becomes row/col header
  - Slice-and-dice (selection and projection) – reduces dimensionality
- Data marts
  - subsets of data on selected subjects
  - e.g. Marketing data mart can include customer, product, sales
  - Department-focused, no enterprise-wide consensus needed
  - But may lead to complex integration problems in the long run
- Relational OLAP (ROLAP)
  - On top of standard relational DBMS
  - Data is stored in relational DBMS
  - Supports extensions to SQL to access multi-dimn. data
- Multidimensional OLAP (MOLAP)
  - Directly stores multidimensional data in special data structures (e.g. arrays)

DW Architecture

- Extract data from multiple operational DB and external sources
- Clean/integrate/transform/store
- refresh periodically
  - update base and derived data
  - admin decides when and how

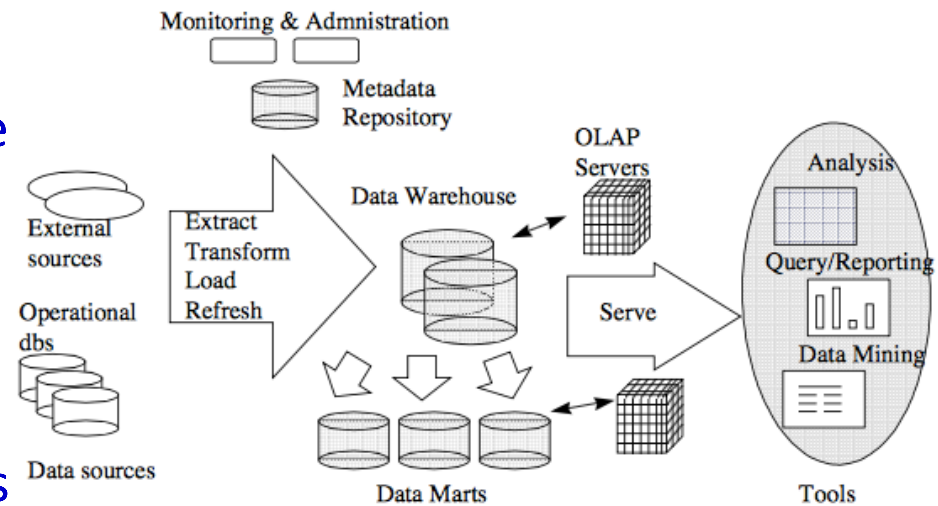


Figure 1. Data Warehousing Architecture

- Main DW and several data marts (possibly)
- Managed by one or more servers and front end tools
- Additional meta data and monitoring/admin tools

ROLAP: Star Schema

- To reflect multi-dimensional views of data
- Single fact table
- Single table for every dimension
- Each tuple in the fact table consists of
  - pointers (foreign key) to each of the dimensions (multi-dimensional coordinates)
  - numeric value for those coordinates
- Each dimension table contains attributes of that dimension

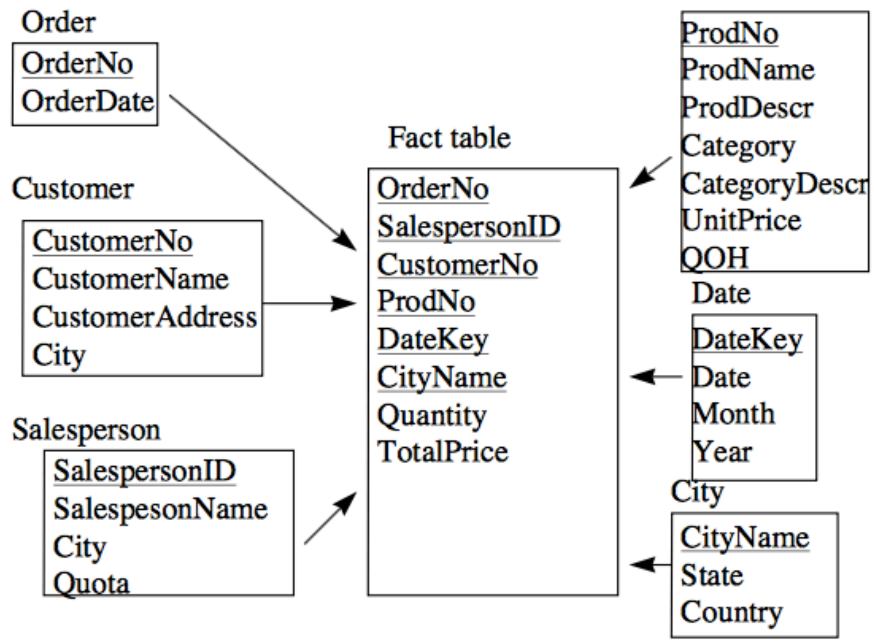


Figure 3. A Star Schema.

No support for attribute hierarchies

ROLAP: Snowflake Schema

- Refines star-schema
- Dimensional hierarchy is explicitly represented

- (+) Dimension tables easier to maintain
  - suppose the "category description is being changed

- (-) Denormalized structure may be easier to browse

- Fact Constellations
  - Multiple fact tables share some dimensional tables
  - e.g. Projected and Actual Expenses may share many dimensions

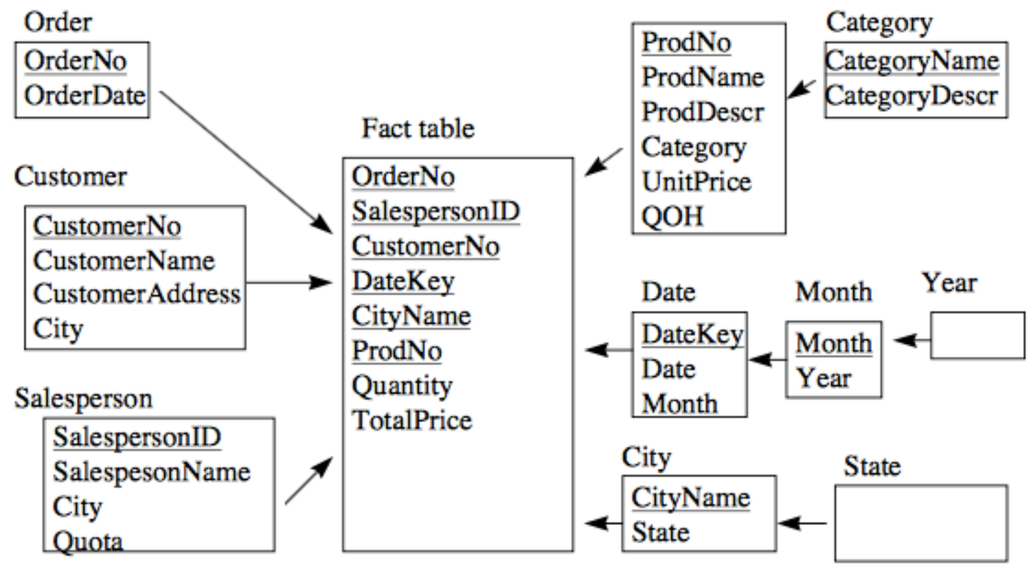


Figure 4. A Snowflake Schema.

Issues to consider

- Index (Lecture 5: Sudeepa)
- Materialization
- Un-nest Queries
- Parallel processing
- Storing meta data

Computing Iceberg Queries Efficiently

Acknowledgement:
Some slides have been taken from Erik Gribkoff's paper presentation, 590q, Winter'14, U. Washington

What is an iceberg query?

SELECT target1, target2, …, targetk, count(rest)
FROM R
GROUPBY target1, target2, …, targetk
HAVING count(rest) >= T

target1	target2	rest
a	e	joe
b	f	fred
a	e	sally
b	d	sally
a	e	bob
c	f	tom

Table 1: Example relation R.

- Computes an aggregate over attributes
- Only output aggregate values above a certain threshold
- Usually, the number of above-threshold results is very small
- The "tip of the iceberg"

- The answer is <a, e, 3> for k = 2, T = 3

Why should we care about Iceberg Queries?

- Many queries in data mining are fundamentally Iceberg queries
- e.g. Market Basket Data Analysis
  - which items are bought together “frequently”
- e.g. find similar documents on web
  - If the number of overlapping chunks >= T
- e.g. Enterprise sales analysis
  - Find the parts-regions pairs where the total sales amount is >= 1M
  - So that the company can order more such parts in those regions

Naïve Approaches

1. Maintain an array of counters in main memory
   - one for each target
   - answer the query in a single pass
   - (-) not always possible – R may not fit in memory
2. Sort R on disk
   - many passes needed to sort
3. Materialization
   - {a, b, c} => [a, b], [a, c], [b, c]
   - A good algorithm uses *virtual* R

- Solutions are "over-kill"
  - do the same amount of work irrespective of the query output size

Iceberg Query Example

LineItem - <partKey, price, numsales, region>

```
CREATE VIEW PopularItems as
    SELECT partKey, region, SUM(numSales * price)
    FROM LineItem
    GROUP BY partKey, region
    HAVING SUM(numSales * price) >= $1,000,000
```

Iceberg Query Example

- Avoiding (near) replicated documents in search engine queries
- Consider table *DocSign* <doc, sig>
  - doc is the document id
  - sig is a signature of a chunk

```
SELECT     D1.doc, D2.doc, COUNT(D1.sig)
FROM       DocSign D1, DocSign D2
WHERE      D1.sig = D2.sig
           AND D1.doc <> D2.doc
GROUP BY D1.doc, D2.doc
HAVING     COUNT( D1.sig) >= T2
```

Document Overlap – Previous Approach

- Broeder et al'97
- Consider table *DocSign* <doc, sig>
  - doc is the document id
  - sig is a signature of a chunk

- Sort *<di, sk> by sk – tuples for a chunk are contiguous*
- for each pair $<d_i,s_k>$ and $<d_j,s_k>$, add $<d_i,d_j>$ to *SignSign*
- sort *SignSign – tuples for a doc are contiguous*
- scan SignSign, count, and check against *T2*
- Case study in the paper:
  - DocSign of size 500MB
  - SignSign size of 40GB
  - although output can only be 1MB !

Terminology

- **R = a materialized relation with <target, rest> pairs**
  - 1 target, 1 rest, for simplicity
- N = |R|
- **V = ordered list of all targets in R**
- V[r] is the r-th most frequent target in R
- n = |V|
- Freq(r) = frequency of V[r] in R

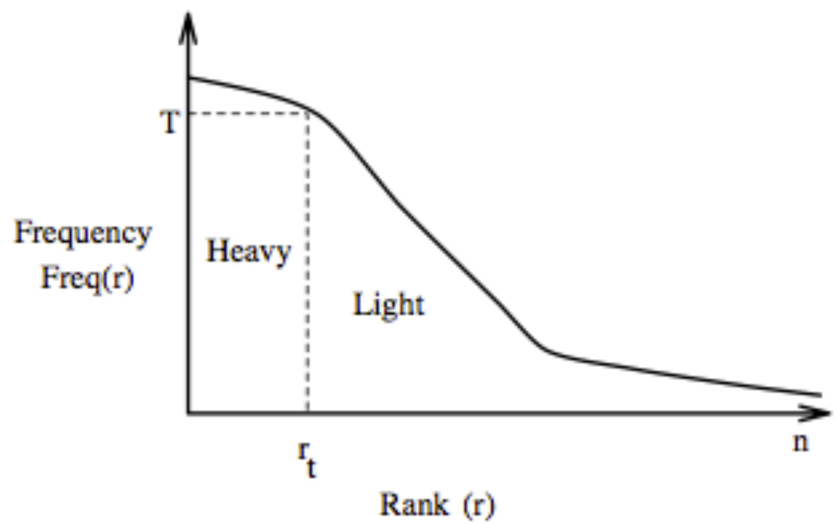


Figure 1: A graphical view of terminology.

Terminology

- **T = threshold**
- r_t = max{ r | Freq(r) >= T}
- H = answer to iceberg query, {V[1], V[2], …, V[r_t]}

- "Heavy targets" – values in H
- The algorithms calculate a "candidate set"

F = potentially heavy targets

- **Goal: F = H**

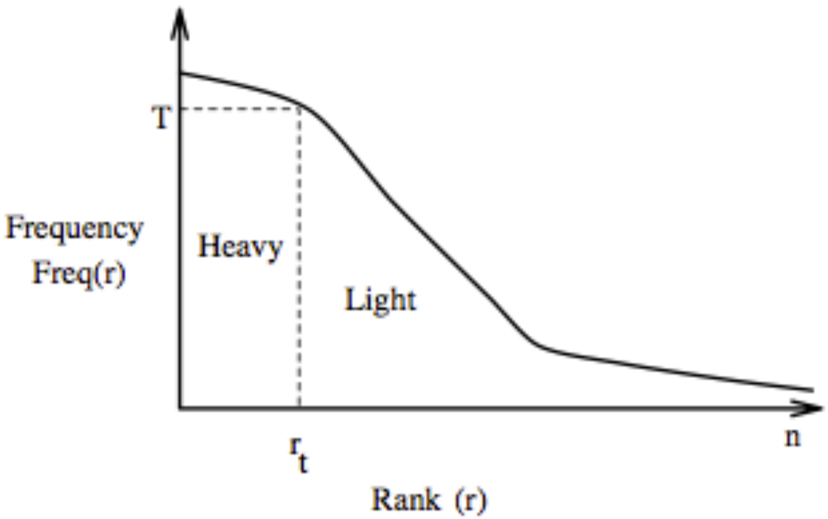


Figure 1: A graphical view of terminology.

False positives and false negatives

- If F – H is non-empty, the algorithm reports false positives
  - If F is small, we can eliminate false positives by counting the frequency of targets in F.
  - As |F| → n, this efficiency deteriorates
  - called COUNT(F)
- If H – F is non-empty, the algorithm generates false negatives
  - Much harder to "regain" in post-processing
  - as hard as original query
  - unless R is highly skewed, i.e. most tuples in R have value from a small set H' = F ∩ H
  - then scan R, eliminate tuples with values in H'
  - run iceberg query to obtain heavy hitters not in H'
- GOAL:
  - Algorithms should have NO False Negatives
  - Algorithms should have AS FEW False Positives AS POSSIBLE

Sampling Algorithm (SCALED-SAMPLING)

- Take a random sample of size s from R
- If the
  - count of a target in the sample
  - scaled by |R|/|s|
  - exceeds T
  - put the target in F

- Pros:
  - Simple
  - Efficient
- Cons:
  - False-positives
  - False-negatives

Coarse-counting algorithm (COARSE-COUNT)

- (not this paper)
- Array A[1…m], Bitmap[1..m]   (m << n = #targets)
- Hash function h: target values → [1…m]

- Perform a linear "hashing" scan of R:
  - For each tuple in R with target v:
    - A[h(v)] += 1
- Set Bitmap [i] = 1 if bucket i is heavy (i.e., A[i] >= T)
- Reclaim memory allocated to A
- "Candidate selection" scan of R:
  - For each target v s.t. Bitmap[h(v)] == 1, add v to F
- Remove false-positives
- Pros:
  - No false-negatives
- Cons:
  - but light elements may be hashed to heavy buckets
    - multiple light elements/ some light some heavy / all heavy
  - F can be large

This paper: Hybrid techniques

- DEFER-COUNT
- MULTI-LEVEL
- MULTI-STAGE

- Combines sampling, multiple hash functions

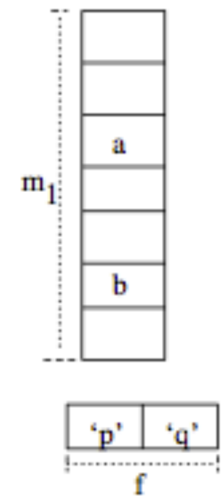
DEFER-COUNT

Idea:

- Use a sampling scan to find initial F
  - small sample s << n (exceeds threshold)
  - add f < s most frequent targets to F (higher prob of being heavy)
- Run hashing-scan exactly the same as COARSE-COUNT, except:
  - Don't increment counters for targets already in F
  - add more targets to F by candidate-selection
  - Remove False Positives from F
  - fewer false positives

Example:

- p, q are heavy targets - identified in sampling phase
  - explicitly maintained in memory, so not counted in buckets
- a, b are light targets
  - hashed values <= T, not counted



(a) DEFER-COUNT

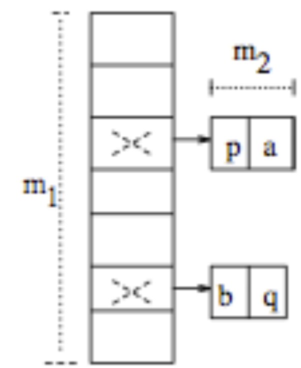
DEFER-COUNT

- Pros:
  - Fewer heavy buckets => fewer false positives
- Cons:
  - Memory split between samples and buckets
  - Maintains explicit targets in memory
  - Have to decide how to choose s and f values
  - If initial F is large, costly to look up each target during hashing scan

MULTI-LEVEL

Sampling Scan:

- Instead of creating an initial F after the sampling scan (s targets)
  - if A[i] >= Ts/n, mark bucket as potentially heavy
  - Allocate m_2 auxiliary buckets
- Reset A counters to 0



(b) MULTI-LEVEL

Hashing Scan

- Increment A[h(v)] if NOT potentially heavy
- Otherwise, hash again into m_2 auxiliary buckets

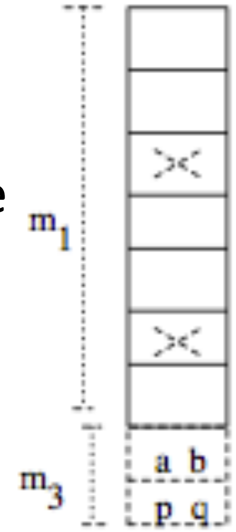
Then count(F)

MULTI-LEVEL

- Pros:
  - does not explicitly maintain the list of potentially heavy targets
    - only maintains counts
    - helps when size of targets is large
- Cons:
  - Still splits memory between primary and auxiliary buckets – how to obtain good split (empirically)
  - Rehashing may be expensive

MULTI-STAGE

- Instead of auxiliary buckets, allocate a common pool of auxiliary buckets B[1,2,…]
  - 50% chance that heavy elements p, q will fall into the same bucket
  - Then no false positives



(c) MULTI-STAGE

- Pros:
  - Makes more efficient use of memory than multi-level
  - fewer false positives (over MULTI-LEVEL)
- Cons:
  - Still splits memory

Optimizing HYBRID with multi-buckets

- Still many light elements may fall into buckets with
  - one or more heavy elements (sampling helps, but not always)
  - many light elements (HYBRID cannot avoid)

- Uniscan
- Multiscan
- Multiscan-shared
- Multiscan-shared2

Described for DEFER-COUNT

- Still do sampling – and store in F – not counted in hashing scan
- Still do COUNT(F) at the end

Single-scan Defer-Count (UNISCAN)

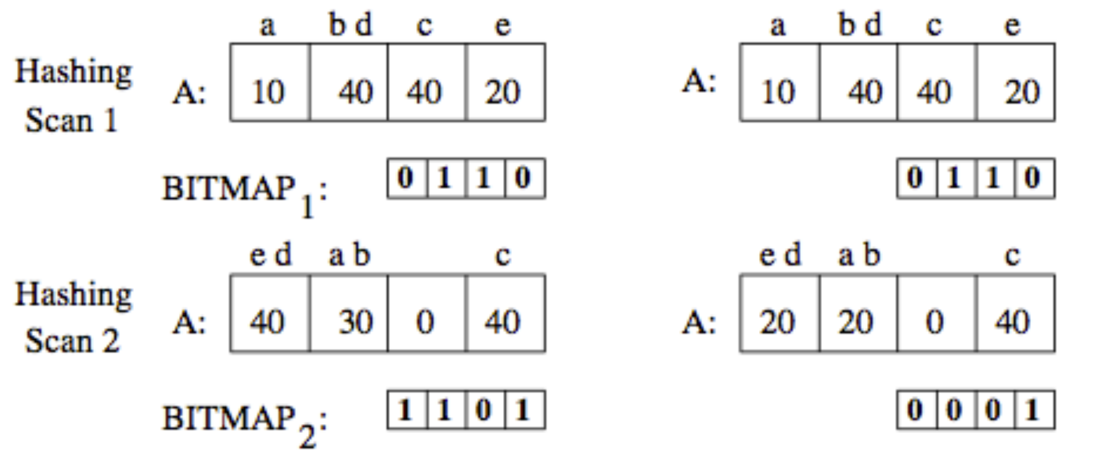
- Idea: Reduce false positives by using additional hash functions.

- Same as defer-count
- but keep k hash functions and bitmaps (smaller space)
- After incrementing counters, add target v to F iff for all k, $BITMAP_k[h_k(v)] = 1$
  - one scan over data

- Choosing k for a given amount of memory is challenging:
  - As k increases, we have many hash tables => fewer false positives
  - As k increases, we also have smaller hash tables => more false positives

MULTISCAN and MULTISCAN-SHARED

- Idea: One hash function per scan
  - then store $BITMAP_k$ on disk
  - then perform next scan.
- read previous k-1 bitmaps from disk to reduce false positives
- MULTISCAN-SHARED: Increment for target only if previous bitmaps say 1
  - e is not counted in the second pass
- MULTISCAN-SHARED2
  - keep hashmaps only from the last q passes
  - fewer bits set to 1, more pruning



(a) MULTISCAN (b) MULTISCAN-SHARED

- a: 10, b: 20, **c: 40**, d: 20, e: 20
- T = 30
- m = 4
- MULTISCAN returns {b, c, d}
- MULTISCAN-SHARED returns {c} - correct

Observations from Case Studies

- Graphs in the paper
- HYBRID
  - MULTI-LEVEL rarely performed well
  - DEFER-COUNT and MULTI-STAGE did well
  - If skew with only a few heavy elements, use DEFER-COUNT with small f (small space in sampling scan)
  - If Data is not too skewed, use MULTI-STAGE (less overhead)
- MULTIBUCKET
  - MULTISCAN-SHARED2 good in general
  - large memory : use UNISCAN

Summary and Conclusions

- Performing multiple passes, helps prune many false positives
- Iceberg queries are found in data-warehousing, data mining etc.
- We saw efficient techniques to execute iceberg queries that are better than conventional schemes