1

Data Warehousing & Association Rule Mining

CPS 296.3: Information Management and Mining

Jun Yang
Duke University
January 20, 2008

† Thanks to contents borrowed from Ullman (http://infolab.stanford.edu/~ullman/mining/mining.html), and Han (http://www.cs.uiuc.edu/homes/hanj/bk2/slidesindex.html)

2

Announcements

- Talk next Tuesday by Irfan Essa from Georgia Tech: ***Computation and Journalism: The Impact of Technology on Journalism, Information Quality, and Civic Literacy***
  - January 27, 4:30-6pm
  - Sanford Institute Room 03
  - References
    - http://www.irfanessa.com/
    - http://www.computational-journalism.com/symposium/index.php
- ➲I strongly encourage you to go—excellent source of project/research ideas

3

Data integration

- Data resides in many distributed, heterogeneous OLTP (On-Line Transaction Processing) sources
  - Sales, inventory, customer, …
  - NC branch, NY branch, CA branch, …
- Need to support OLAP (On-Line Analytical Processing) over an integrated view of the data
- Possible approaches to integration
  - Eager: integrate in advance and store the integrated data at a central repository called the data warehouse
  - Lazy: integrate on demand; process queries over distributed sources—mediated or federated systems

4

Eager vs. lazy integration

Eager (warehousing)
- In advance: before queries
- Copy data from sources
- Answer could be stale
- Need to maintain source-warehouse consistency
- Query processing is local to the warehouse
  - Faster
  - Can operate when sources are unavailable

Lazy
- On demand: at query time
- Leave data at sources
- Answer is more up-to-date
- No need to maintain consistency of copies
- Sources participate in query processing
  - Potentially slower
  - Interferes with local processing

5

Maintaining a data warehouse

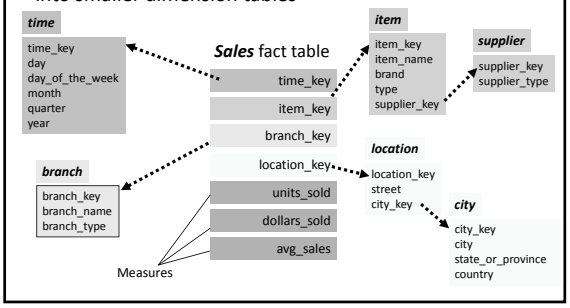
- The "ETL" process
  - Extract relevant data and/or changes from sources
  - Transform data/changes to match the warehouse schema
  - Load (and integrate) data/changes into the warehouse
- Approaches to maintenance
  - Recomputation
    - Take periodic dumps of sources, say, every night
    - What if there is no "night"?
    - What if recomputation takes more than a day?
  - Incremental maintenance
    - Compute and apply only incremental changes
    - Not easy for complicated transformations
    - Fast if changes are small
    - Need to detect incremental changes at the sources

6

"Star" warehouse schema

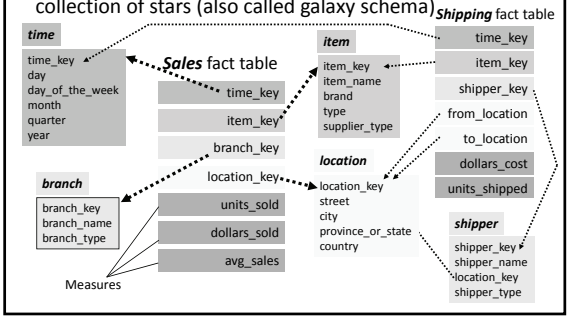
Beyond ★: snowflakes

- Some dimensions contain hierarchies that are normalized into smaller dimension tables

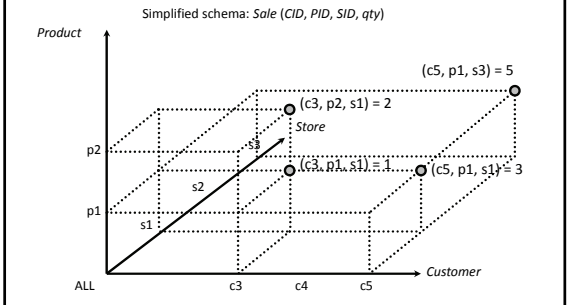


Beyond ★: fact constellations

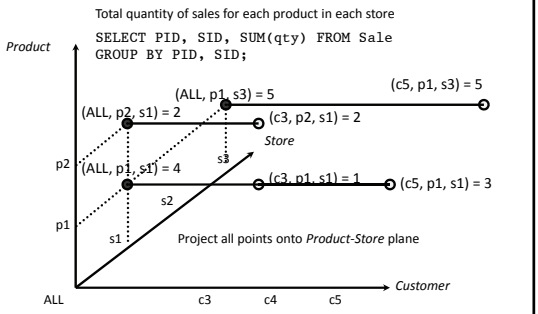
- Multiple fact tables share dimension tables, viewed as a collection of stars (also called galaxy schema)



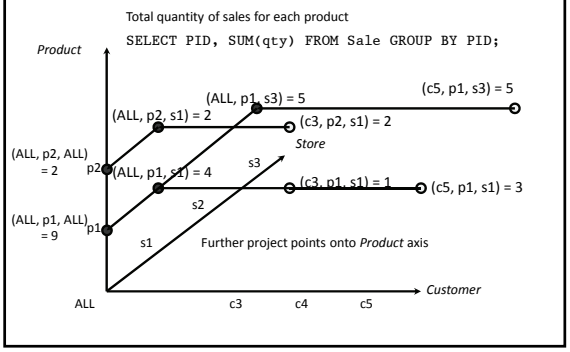
Data cube



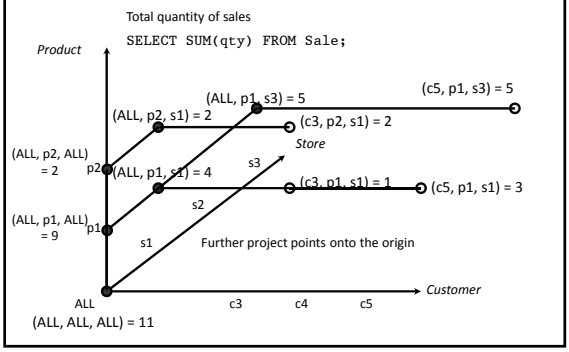
Completing the cube — plane



Completing the cube — axis



Completing the cube — origin



CUBE operator

- *Sale* (*CID*, *PID*, *SID*, *qty*)
- Proposed SQL extension:
  ```
  SELECT SUM(qty) FROM Sale
  GROUP BY CUBE CID, PID, SID;
  ```
- Output contains the union of:
  - Normal groups produced by GROUP BY
    - (c1, p1, s1, sum), (c1, p2, s3, sum), etc.
  - Groups with one or more ALL's
    - (ALL, p1, s1, sum), (c2, ALL, ALL, sum), (ALL, ALL, ALL, sum), etc.

Gray et al., "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total." *ICDE* 1996

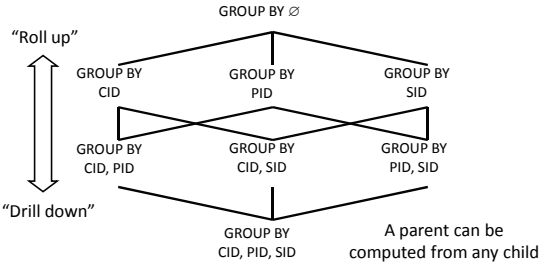
Materialized views

- Computing GROUP BY and CUBE aggregates is expensive
- OLAP workloads perform these operations over and over again

☞An effective solution
- Precompute and store the aggregates as materialized views
- Maintain them automatically as base data changes

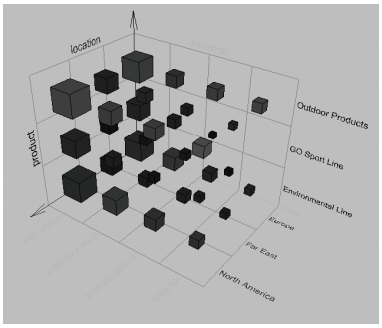
Aggregation view lattice

Also called cuboids in a data cube



A parent can be computed from any child

Exploring a data cube



Selecting views to materialize

- Factors in deciding what to materialize
  - What is its storage cost?
  - What is its update cost?
  - Which queries can benefit from it?
  - How much can a query benefit from it?
- Example
  - GROUP BY ∅ is small, but not useful to most queries
  - GROUP BY CID, PID, SID is useful to any query, but too large to be beneficial

Harinarayan et al., "Implementing Data Cubes Efficiently." *SIGMOD* 1996

Frequent itemset mining

- Market-basket model
  - A large set of items
    - E.g., things sold in a supermarket
  - A large set of baskets, each containing a small set of the items
    - E.g., things one customer buys in one shopping trip
- Find all frequent itemsets
  - Support of itemset X = $\frac{\text{\# of baskets containing all of } X}{\text{total \# of baskets}}$
  - X is frequent if its support $\geq$ a prescribed support threshold
  - Examples: {diaper, beer}, …

TID	*items*
T001	diaper, milk, candy
T002	milk, egg
T003	milk, beer
T004	diaper, milk, egg
T005	diaper, beer
T006	milk, beer
T007	diaper, beer
T008	diaper, milk, beer, candy
T009	diaper, milk, beer
…	…

Applications

- Real market baskets: what do customers buy together?
  - Position tempting items within stores
  - Run tie-in "tricks": sale on diapers while raising beer price
  - ➲High support needed to make it worthwhile
- Linguistic analysis: items = words; baskets = paragraphs
  - Find words that appear together unusually frequently, e.g., linked concepts
- Biomedical research: items = genes, blood-chemistry factors, diseases; baskets = people
  - Detect factors that frequently show up with diseases
    - ➲ Obviously you will need more than frequent itemsets for conclusions!

Association rules

- If-then rules about the contents of baskets
- Given itemsets X and Y, $X \rightarrow Y$ means
  - If a basket contains X then it is likely to contain Y
- More precisely
  - Support of the rule = support($X \cup Y$)
  - Confidence of the rule = support($X \cup Y$) / support(X)
    - I.e., conditional probability that a basket also contains Y given that it contains X
- Association rule mining: find all association rules with support $\geq s$ and confidence $\geq c$

➲Far from implying any causality!

Computational model

- Scale is enormous
  - WalMart has ~100k items and billions of baskets
  - Web has ~100M words and billions of pages
- Typically, you have to sequentially scan data in flat files
- I/O-model of computation
  - Count the # of disk I/Os (in blocks)
    - In this case, # of passes is a reasonable measure
  - Keep an eye on memory requirement—next slide
- Computationally, the major task is finding frequent itemsets—focus for the rest of this lecture

Naïve algorithm

One pass!

- Keep a running count for each possible itemset
- For each basket, and for each itemset X contained in the basket, increment the count for X
- Return itemsets with support $\geq s$

Problem?

- # of itemset is huge: 2^n, where n = # of items

Think: how do we prune the search space?

The a-priori property

- *All subsets of a frequent itemset must also be frequent*
  - Because any basket that contains X must also contain subsets of X

➲If X has been verified to be infrequent, there is no need to count X's supersets because they must be infrequent too!

Apriori algorithm [Agrawal & Srikant VLDB '94]

Multiple passes over the transactions

- Start with k = 1
- Pass k finds all frequent k-itemsets (i.e., itemset of size k)
  - Keep one counter for each
    - ➲ BTW, for each basket, how do you increment appropriate counters efficiently?
- Use the set of frequent k-itemsets found in pass k to construct candidate (k+1)-itemsets to be counted in pass (k+1)
  - A (k+1)-itemset is a candidate only if all its subsets of size k are frequent

Example: pass 1

TID	items
T001	A, B, E
T002	B, D
T003	B, C
T004	A, B, D
T005	A, C
T006	B, C
T007	A, C
T008	A, B, C, E
T009	A, B, C
T010	F

Baskets
s = 20%

itemset	count
{A}	6
{B}	7
{C}	6
{D}	2
{E}	2

Frequent 1-itemsets
(Itemset {F} is infrequent)

Example: pass 2

TID	items
T001	A, B, E
T002	B, D
T003	B, C
T004	A, B, D
T005	A, C
T006	B, C
T007	A, C
T008	A, B, C, E
T009	A, B, C
T010	F

Baskets
s = 20%

Generate candidates → Scan and count → Check min. support

itemset	count
{A}	6
{B}	7
{C}	6
{D}	2
{E}	2

Frequent 1-itemsets

itemset	count
{A,B}	4
{A,C}	4
{A,D}	1
{A,E}	2
{B,C}	4
{B,D}	2
{B,E}	2
{C,D}	0
{C,E}	1
{D,E}	0

Candidate 2-itemsets

itemset	count
{A,B}	4
{A,C}	4
{A,E}	2
{B,C}	4
{B,D}	2
{B,E}	2

Frequent 2-itemsets

Example: pass 3

TID	items
T001	A, B, E
T002	B, D
T003	B, C
T004	A, B, D
T005	A, C
T006	B, C
T007	A, C
T008	A, B, C, E
T009	A, B, C
T010	F

Baskets
s = 20%

Generate candidates → Scan and count → Check min. support

itemset	count
{A,B}	4
{A,C}	4
{A,E}	2
{B,C}	4
{B,D}	2
{B,E}	2

Frequent 2-itemsets

itemset	count
{A,B,C}	2
{A,B,E}	2

Candidate 3-itemsets

itemset	count
{A,B,C}	2
{A,B,E}	2

Frequent 3-itemsets

Example: pass 4

TID	items
T001	A, B, E
T002	B, D
T003	B, C
T004	A, B, D
T005	A, C
T006	B, C
T007	A, C
T008	A, B, C, E
T009	A, B, C
T010	F

Baskets
s = 20%

Generate candidates

itemset	count
{A,B,C}	2
{A,B,E}	2

Frequent 3-itemsets

Candidate 4-itemsets

No more itemsets to count!

Example: final answer

itemset	count
{A}	6
{B}	7
{C}	6
{D}	2
{E}	2

Frequent 1-itemsets

itemset	count
{A,B}	4
{A,C}	4
{A,E}	2
{B,C}	4
{B,D}	2
{B,E}	2

Frequent 2-itemsets

itemset	count
{A,B,C}	2
{A,B,E}	2

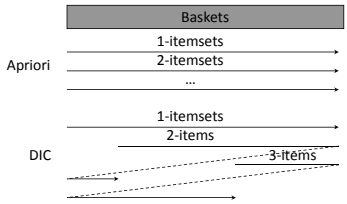
Frequent 3-itemsets

Quest for better algorithms

- Do more with each pass
- Sampling
- Progressively reduce problem size
- …

DIC [Brin et al. SIGMOD '97]

- Once all size-(k – 1) subsets of a k-itemset X are known to be frequent, we immediately begin counting X in the same pass
  - May need to continue counting X in the next pass



PCY [Park, Chen, Yu SIGMOD '05]

- Observation: memory usage of a-priori is very uneven
  - During Pass 1, most memory is idle!
  - ➲Use the memory to count 2-itemsets?
- Too many to count? Count roughly
- Hash itemsets into a fixed # of buckets
  - # determined by available memory
- One counter per bucket, shared by the itemsets hashed to it
  - If counter < s, no itemset in the bucket is frequent
  - But if counter $\geq s$, we cannot say much
    - Can happen even if all itemsets in the bucket are infrequent!
- In Pass 2, only count a 2-itemset if all its size-1 subsets are frequent, and if it hashes to a frequent bucket

PCY extensions

- Use two independent hash functions & tables
  - Pro: an itemset can only be frequent if both of its buckets are
  - Con: # of itemset/bucket doubled, so buckets are more likely to be frequent
- Or, make another pass with an independent hash
  - Pro: this pass can have a lot of buckets too
  - Con: another pass, and still need to remember a frequent bucket bitmap from the last pass

Mine a sample?

- Get a sample $\mathcal{S}$ of all baskets, and run Apriori on it
  - Hopefully $\mathcal{S}$ is small enough for memory, so no I/Os
- What can you say about frequents itemsets in $\mathcal{S}$ vs. those in the original dataset?
  - If an itemset is (in)frequent $\mathcal{S}$, it is more likely to be (in)frequent in the original dataset as well
    - No guarantee though

SON [Savasere, Omiecinski, Navathe, VLDB '95]

Pass 1

- Partition the dataset, and mine each partition

➲An itemset cannot be frequent unless it is frequent in at least one partition

Pass 2

- Count every candidate itemset—one that was found to be frequent in at least one partition

Pass 2, distributed version: each partition owned by a node

- Nodes distribute locally frequent itemsets to all others
- Each node counts all candidate itemsets locally
- Nodes aggregate their counts

Toivonen [VLDB '96]

- Mine a sample, using a slightly lower support threshold
  - Reduces the chance of missing a frequent itemset
- Make one full pass, and count
  - All itemsets frequent in the sample
  - Their negative border: an itemset X is in the negative border if X is infrequent but all its subsets are frequent
- If all itemsets in the negative border turn out infrequent in the full pass, then complete answer = frequent itemsets found in the full pass

  ➲Why?
- Otherwise, start over

FP-growth [Han et al. SIGMOD '00]

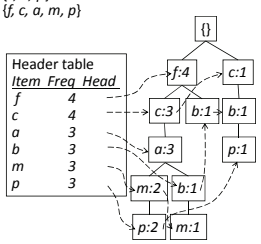
- Observations
  - Too much scanning of irrelevant data past Pass 1
  - Too much redundant counting work
- Ideas
  - Compress the dataset into a smaller structure with just relevant information
  - Organize the information in a way to facilitate result reuse

Constructing FP-tree

TID	Items bought	(ordered) frequent items
100	{f, a, c, d, g, i, m, p}	{f, c, a, m, p}
200	{a, b, c, f, l, m, o}	{f, c, a, b, m}
300	{b, f, h, j, o, w}	{f, b}
400	{b, c, k, s, p}	{c, b, p}
500	{a, f, c, e, l, p, m, n}	{f, c, a, m, p}

min_support = 3

- Make one pass to find frequent items
- Sort them in order of decreasing frequency
- Make another pass to build FP-tree
  - Like a trie; keep counts (of prefixes) at nodes
  - Link tree nodes for the same item in a list



The FP-growth strategy

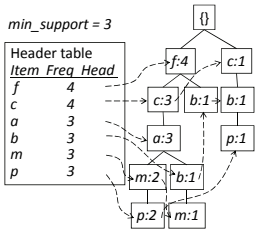
- Sorted list of frequent items: f-c-a-b-m-p
- Partition potentially frequent itemsets as follows
  - Those containing p
  - Those containing m but not p
  - …
  - Those containing c but not a, b, m, or p: {c}, {f, c}
  - {f}

➲Mine these partitions separately

➲Mining of each partition starts with a conditional FP-tree and proceeds recursively

Building cond. FP-trees (1)

- Traverse the list for each frequent item and accumulate its transformed prefix paths to form its conditional pattern base

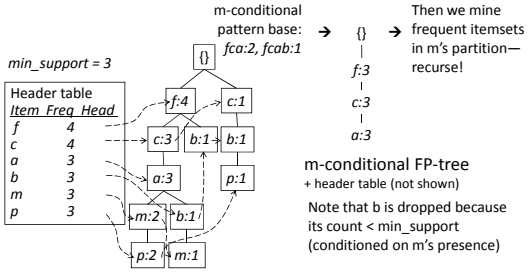


Item	Cond. pattern base
c	f:3
a	fc:3
b	fca:1, f:1, c:1
m	fca:2, fcab:1
p	fcam:2, cb:1

"m-conditional" means we only look at those baskets containing item m

Building cond. FP-trees (2)

- Build conditional FP-tree from the conditional pattern base

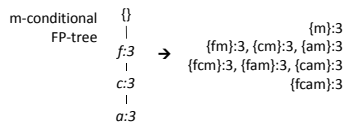


m-conditional FP-tree
+ header table (not shown)

Note that b is dropped because its count < min_support (conditioned on m's presence)

Recursing on FP-trees

- If you see a single-path X-conditional FP-tree, return all combinations of items in this path, each unioned with X



- Otherwise continue: for each item x in header table, build an $(X \cup \{x\})$-conditional FP-tree and mine it

Summary

- Association rule mining—arguably the one problem that started it all
  - The two early papers R. Agrawal in 1993-1994 have more than 12k citations according to Google Scholar as of Jan. 2009!
- What is the difference on focus from machine learning and statistics?
  - Simple in problem formulation
  - Driven by scalability and performance