

Data Warehousing

Jens Teubner, TU Dortmund
jens.teubner@cs.tu-dortmund.de

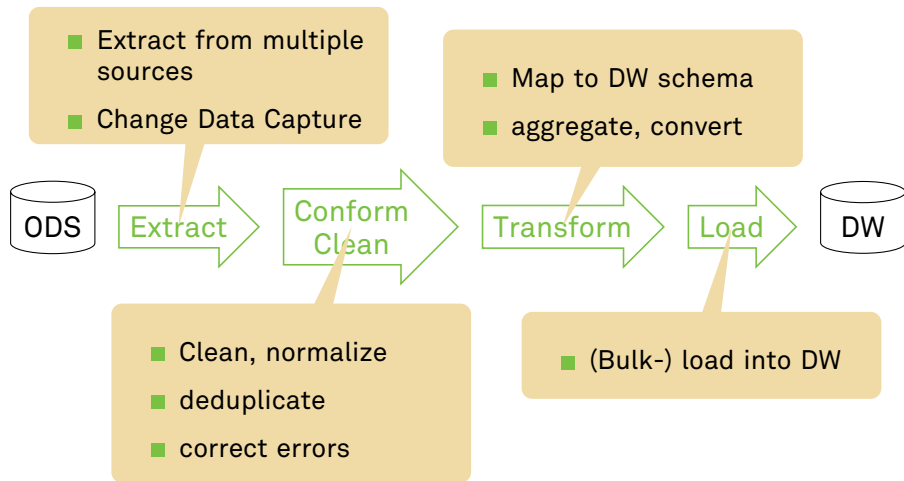
Winter 2014/15

Part VI

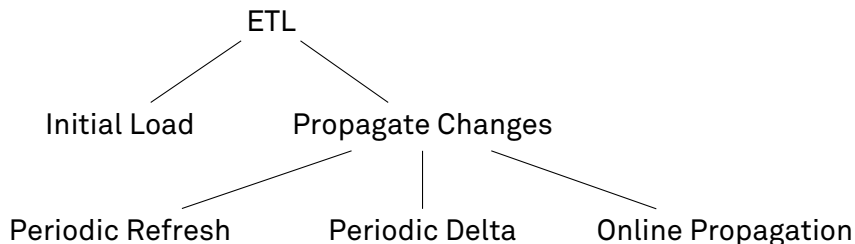
ETL Process

ETL Overview

In most DW systems, the most complex part is the **ETL process**.



When do we run the ETL process?





Considerations:

- **Overhead** on data warehouse and source sides.
 - *E.g.*, online propagation puts a permanent burden on both sides; cannot benefit from **bulk loading** mechanisms
- **Data Staleness**
 - Frequent updates reduce staleness, but increase overhead.
- **Debugging, Failure Handling**
 - With online/stream-based mechanisms, it may be more difficult to track down problems.
- Different process for different flavors of data?
 - *E.g.*, periodic refresh may work well for small (dimension) tables.

Detecting changes is a challenge:

- **Audit Columns** (e.g., “last modified” time stamp)

- Set time stamps or “new” flags on every row update  **How?**
- Unset “new” flags on every load into the DW.  **Why?**

- **Full Diff**

- Keep old snapshot and diff it with current version.
- Thorough, will detect any change
- Resource-intensive: need to **move** and **scan** large volumes
- Optimization: Hashes/checksums to speed up comparison

- **Database Log Scraping**

- The database’s write-ahead log contains all change inform.
- Scraping the log may get messy, though.
- Variant: create a **message stream** ODS → DW

Data Cleansing

After extraction, data has to be **normalized** and **cleaned**.

	Name	Street	City	Phone
r_1	Sweetlegal Investments Inc	202 North	Redmond	425-444-5555
r_2	ABC Groceries Corp	Amphitheatre Pkwy	Mountain View	4081112222
r_3	Cable television services	One Oxford Dr	Cambridge	617-123-4567

	Name	Street	City	Phone
s_1	Sweet legal Invesments Inc.	202 N	Redmond	
s_2	ABC Groceries Corpn.	Amphitheetre Parkway	Mountain View	
s_3	Cable Services	One Oxford Dr	Cambridge	6171234567

Problem:

- Real-world data is **messy**.
- Consistency rules in the OLTP system?
 - A lot of data is still entered by people.
 - Data warehouses serve as an **integration platform**.

Typical cleaning and normalization tasks:

- Correct **spelling errors**.
- Identify **record matches** and **duplicates**.
- Resolve **conflicts** and **inconsistencies**.
- **Normalize** (“conform”) data.

1 Similarity Join

- Bring together similar data
- For record matching and deduplication

2 Clustering

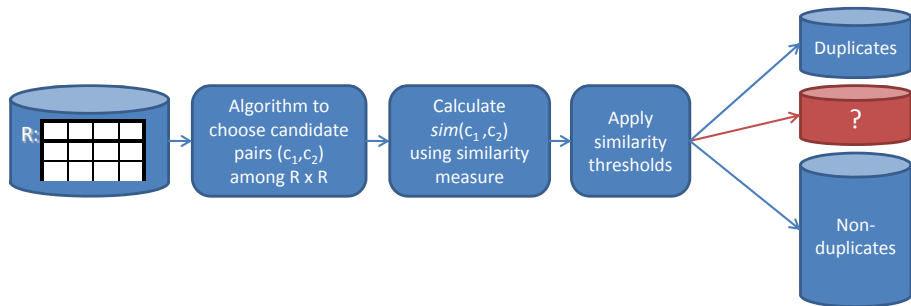
- Put items into groups, based on “similarity”
- *E.g.*, pre-processing for deduplication

3 Parsing

- *E.g.*, source table has an ‘address’ column; whereas target table has ‘street’, ‘zip’, and ‘city’ columns
- Might have to identify pieces of a string to normalize (e.g., "Road" → "Rd")

Similarity Join / Deduplication

Process of finding duplicates:



What is the “similarity” of two strings s_1 and s_2 ?

1 Edit Distance

$ed(s_1, s_2)$: shortest **edit sequence** that transforms s_1 into s_2

insert $ab \rightarrow axb$


delete $axb \rightarrow ab$

replace $axb \rightarrow ayb$

transpose $axyb \rightarrow ayxb$

E.g. $s_1 = \text{"Sweet"}; s_2 = \text{"Sweat"}$

→ Levenshtein distance (insert, delete, replace allowed): 

→ Longest Common Subsequence (LCS) distance (insert, delete allowed) 

2 Jaccard Similarity

Intuition: similarity of two **sets** S_1 and S_2

$$\rightarrow \frac{\text{size of intersection}}{\text{size of union}} = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

Sets? String $s_i \rightarrow$ set S_i ?

Trick: Determine q -grams of s_i

\rightarrow q -gram: all substrings of size q

\rightarrow $qgrams("Sweet") = \{Sw, we, ee, et\}$

$$\begin{aligned} \rightarrow \frac{|\{Sw, we, ee, et\} \cap \{Sw, we, ea, at\}|}{|\{Sw, we, ee, et\} \cup \{Sw, we, ea, at\}|} &= \frac{|\{Sw, we\}|}{|\{Sw, we, ee, et, ea, at\}|} \\ &= \frac{2}{6} = \frac{1}{3} \end{aligned}$$

3 Soundex

Phonetic algorithm to index words by sound:

1. Retain first letter of word.
2. Replace following letters with numbers (drop other letters):

b, f, p, v → 1

c, g, j, k, q, s, x, z → 2

d, t → 3

l → 4

m, n → 5


r → 6

3. Drop letters where preceding letter yielded same number.
4. Collect three numbers, fill with 0 if necessary.

→ *soundex*("Sweet") = S300;


soundex("Robert") = *soundex*("Rupert") = R163.

Similarity Join—Naïve Strategy

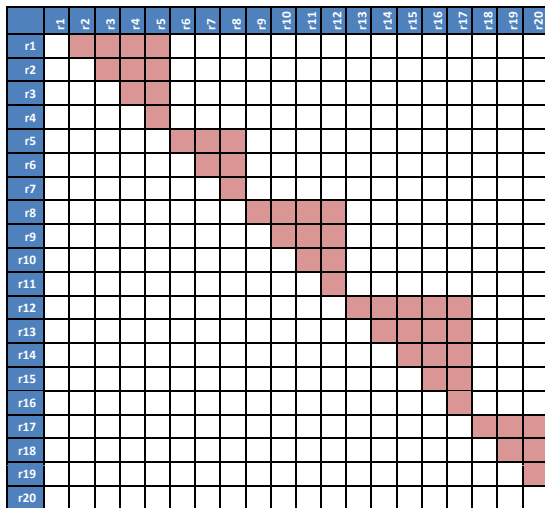
- Compare every record with every other
- Here: Deduplication ($R \bowtie_{\approx} R$)
-  **Cost?**

	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11	r12	r13	r14	r15	r16	r17	r18	r19	r20
r1																				
r2																				
r3																				
r4																				
r5																				
r6																				
r7																				
r8																				
r9																				
r10																				
r11																				
r12																				
r13																				
r14																				
r15																				
r16																				
r17																				
r18																				
r19																				
r20																				

Similarity Join—Blocking

- Partition data into **blocks**
- Compare only **within** blocks
-  **Cost?**
Assume n records and b blocks:

$$\begin{aligned} b \cdot \frac{n}{b} \left(\frac{n}{b} - 1 \right) \\ &= \frac{n \left(\frac{n}{b} - 1 \right)}{2} \\ &= \frac{1}{2} \left(\frac{n^2}{b} - n \right) \end{aligned}$$



	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11	r12	r13	r14	r15	r16	r17	r18	r19	r20
r1																				
r2																				
r3																				
r4																				
r5																				
r6																				
r7																				
r8																				
r9																				
r10																				
r11																				
r12																				
r13																				
r14																				
r15																				
r16																				
r17																				
r18																				
r19																				
r20																				

Observations:

- Must partition such that duplicates appear in same partition.
 - Risk of **missing** duplicates.

Strategies:

- Use (prefix of) the ZIP code
 - Assume no typo in the ZIP code and customer has not moved across ZIP code ranges.
- Use first character of last name
 - Again, assume no typo there.



Typically leads to **uneven partition sizes**.

Refinement:

- Run **multiple passes** of the similarity join.
 - Use **different partitioning** key in each pass.
 - Assume duplicates agree in at least one partitioning key.

Similarity Join—Sorted Neighborhood

- 1 Assign a **sort key** to each record.
- 2 **Sort** accordingly.
- 3 **Slide window** of size w across sorted list and join within.

Number of comparisons:

$$(w - 1) \cdot \left(n - \frac{w}{2}\right)$$

	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11	r12	r13	r14	r15	r16	r17	r18	r19	r20
r1		■	■	■																
r2			■	■	■															
r3				■	■	■														
r4					■	■	■													
r5						■	■	■												
r6							■	■	■											
r7								■	■	■										
r8									■	■	■									
r9										■	■	■								
r10											■	■	■							
r11												■	■	■						
r12													■	■	■					
r13														■	■	■				
r14															■	■	■			
r15																■	■	■		
r16																	■	■	■	
r17																		■	■	■
r18																			■	■
r19																				■
r20																				

Similarity Join—Sorted Neighborhood

Observations:

- Need good sorting criterion
 - Choose characters with low probability of errors

Example:

- Sort by
 - First 3 consonants of last name
 - First letter of last name
 - First 2 digits of ZIP code.

(It is more likely to err in a vowel than in a consonant.)

Also:

- **Multi-pass processing** can be beneficial also here.

Data screening system:

- 1 Column screens:** Test data within a column
 - Correct value ranges, value formatting, null values?
- 2 Structure screens:** Relationship across columns
 - Foreign key relationships?
 - Combination of columns is a valid postal address?
- 3 Business rule screens:** Data plausible according to business rules?
 - *E.g.*, customer status X requires N years of loyalty, M EUR total revenue, etc.

Cleansing: Tool Support

Lots of **tools** support typical cleaning tasks:

- Commercial offerings:
 - SAP Business Objects
 - IBM InfoSphere Data Stage
 - Oracle Data Quality and Oracle Data Profiling
- Open source tools:
 - Eobjects DataCleaner, Talend Open Profiler
- Explore and profile source data
 - Analyze key properties, missing values, distributions, etc.
- Rules for common filtering and normalization tasks
 - Regular expressions for phone numbers, credit card numbers, etc.
 - Convert dates, phone numbers, addresses, etc.

Tools also help with **schema integration**.

- Different source systems, types, and schemas must be integrated.
- Infer **mapping** between schemas (automatically)?

Tools:

- Compare table and attribute names; consider synonyms and homonyms
- Infer data types/formats and mapping rules
- ↪ Techniques similar to **similarity joins** and **deduplication**.

Still:

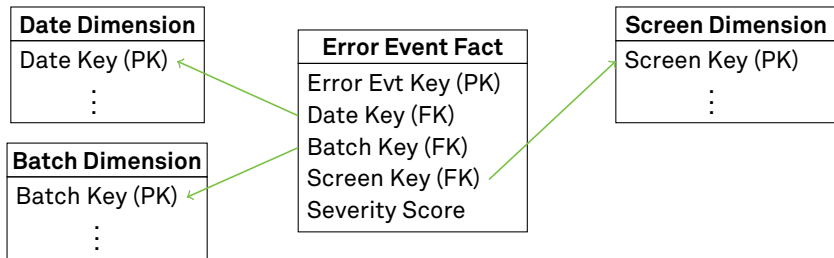
- Often a lot of **manual work** needed.

What to do with detected errors/problems?

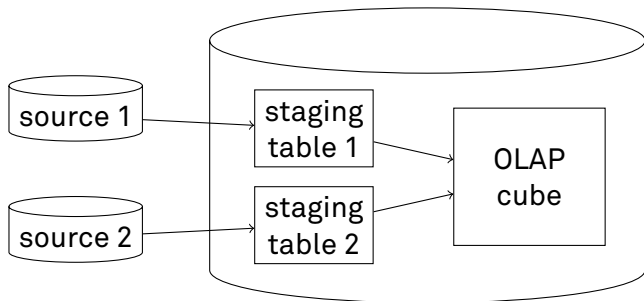
- Fix automatically if possible
- Otherwise: **report the error** → **How/Where?**

“Trick:” Error event schema

→ Star schema for the “error” business event



Advantage?



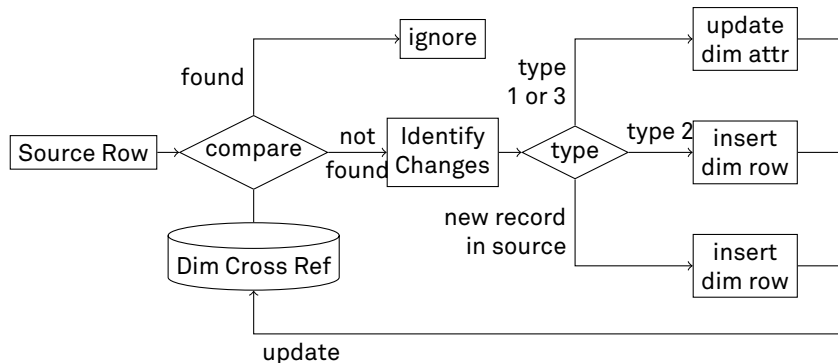
Source → Staging Table:

- Tool depends on data source (database, XML, flat files, etc.)
 - e.g., SQL, XQuery, Perl, awk, etc.
- Often:
 - Extract to flat file (e.g., CSV)
 - Then **bulk-load** into staging table

Prepare Dimension Tables

Check...

- ...whether dimension row is new
- ...which fields were changed
- ...which type of update handling was intended (↗ slides 68ff.)



“update else insert”, “insert if not exists”, etc.

- often real **performance killers**
- better: separate updates and **bulk-load** inserts

Must re-compute aggregates (Type 1 updates)

- again, **bulk-loading/changing** is your friend

Generate and find dimension surrogate keys

- e.g., use key generator of back-end DB
- Maintain “Dim Cross Ref” table in memory or in back-end DB

Tricks to load data **fast**:

1 Turn off logging

- Databases maintain a **write-ahead log** to implement **failure tolerance** mechanisms.
- Row-by-row logging causes huge **overhead**.

2 Pre-sort data

- Depending on system, may speed up **index construction**.
- Additional benefit: may result in better **physical layout**

3 Truncate table first

- Makes (not) logging and failure tolerance even easier.

4 Enable “fast mode”

→ If data is **prepared properly**, database may use **faster parsing mechanisms**

5 Make sure data is correct

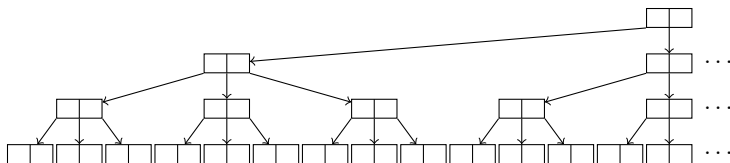
→ Transformation, field truncation, error reporting may slow down bulk-loading significantly

6 Temporarily disable integrity control

→ Avoid checking during load, but do it in bulk, too.

Example: Bulk Loading and B-Tree Indexes

Building a B⁺-tree is particularly easy when the input is **sorted**.



- Build B⁺-tree **bottom-up** and **left-to-right**.
- Create a parent for every $2d + 1$ unparented nodes.
- If data is not sorted already, database will typically sort it before loading/re-building the index.