# Data Warehousing and Data Mining

Introduction to Databases

CompSci 316 Spring 2017

DUKE
COMPUTER SCIENCE

# 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

# OLTP versus OLAP

OLTP

- Mostly updates
- Short, simple transactions
- Clerical users
- Goal: transaction throughput

OLAP

- Mostly reads
- Long, complex queries
- Analysts, decision makers
- Goal: fast queries

Implications on database design and optimization?

OLAP databases do not care much about redundancy

- "Denormalize" tables
- Many, many indexes
- Precomputed query results

# Eager versus lazy integration

Eager (warehousing)

- In advance: before queries
- Copy data from sources

☞Answer could be stale

☞Need to maintain 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

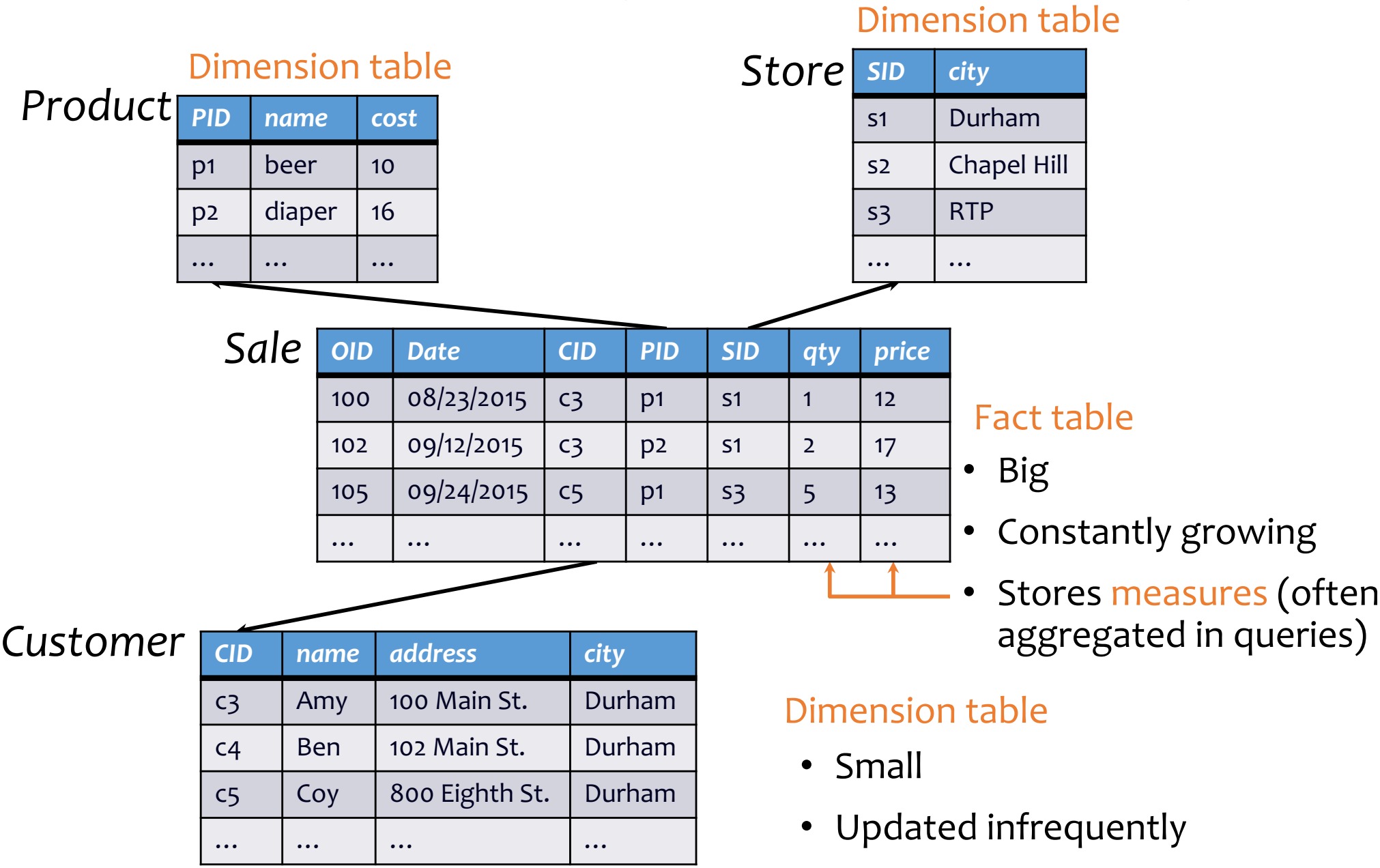
☞No need to maintain consistency

☞Sources participate in query processing
  - Slower
  - Interferes with local processing
  - Still has consistency issues

# Maintaining a data warehouse

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

# "Star" schema of a data warehouse



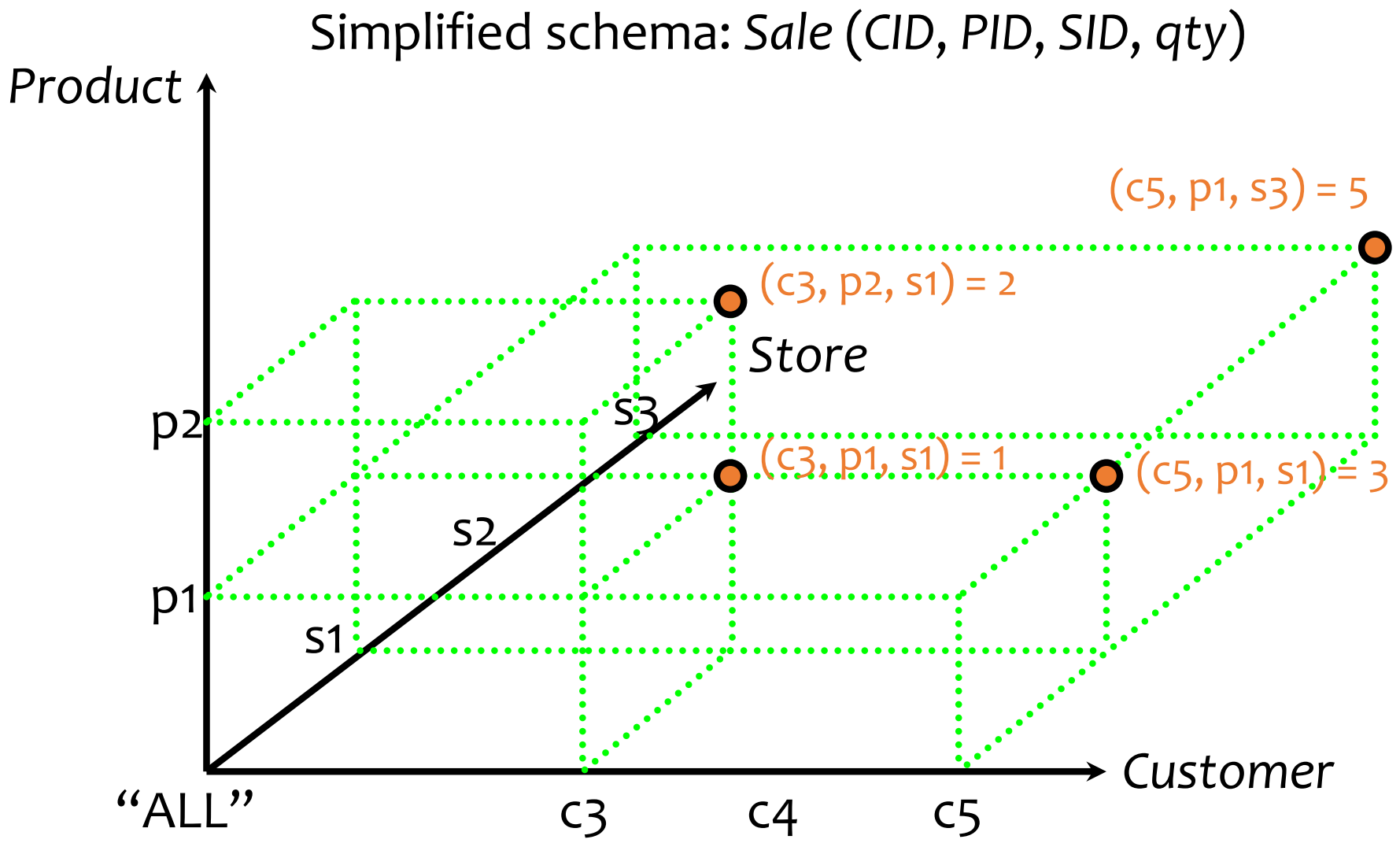

Fact table

- Big
- Constantly growing
- Stores measures (often aggregated in queries)

Dimension table

- Small
- Updated infrequently

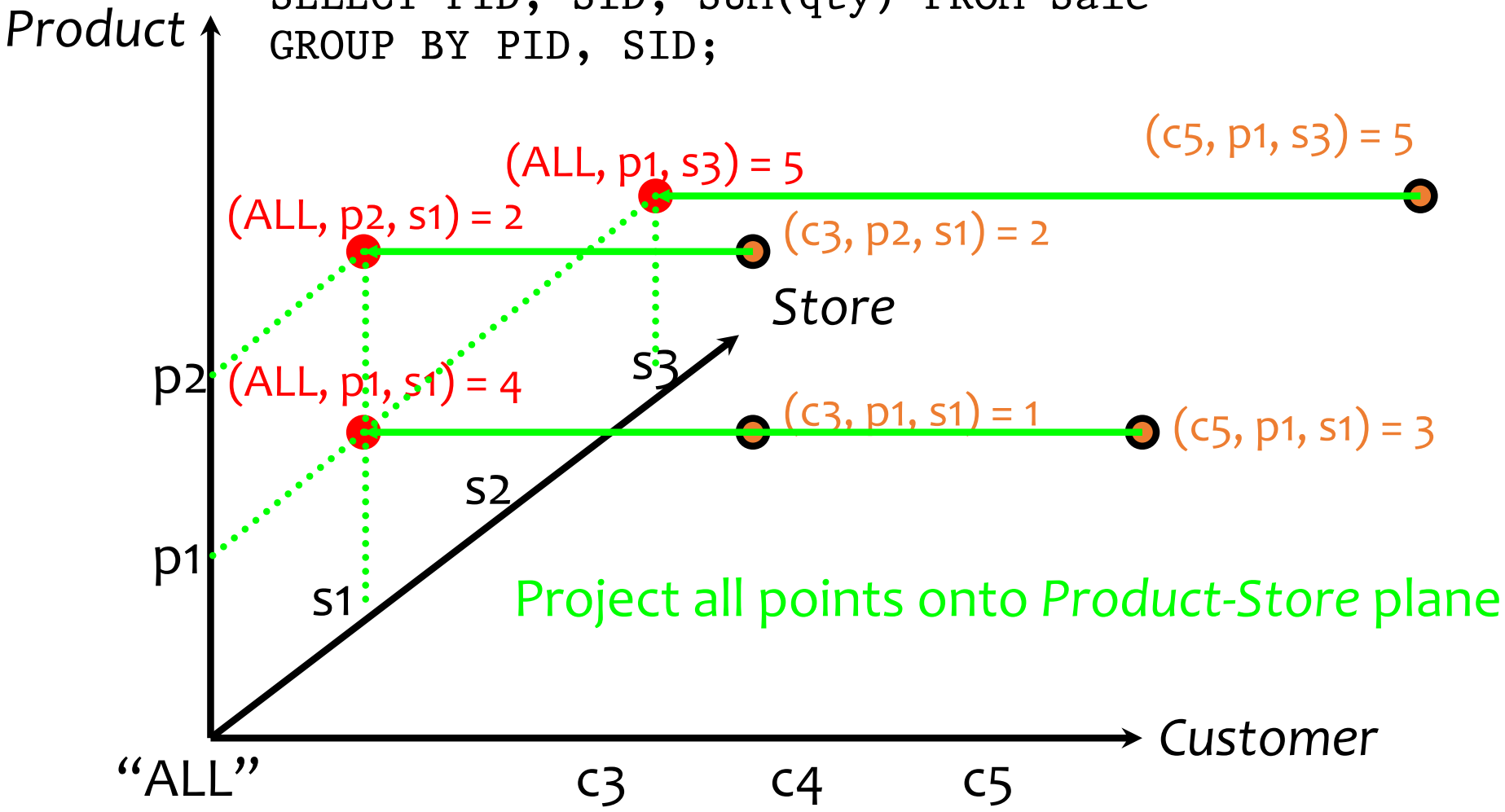
# Data cube


Simplified schema: Sale (CID, PID, SID, qty)
Product
(c5, p1, s3) = 5
(c3, p2, s1) = 2
Store
p2
s3
(c3, p1, s1) = 1
(c5, p1, s1) = 3
s2
p1
s1
Customer
"ALL"
c3
c4
c5

# Completing the cube—plane

Total quantity of sales for each product in each store

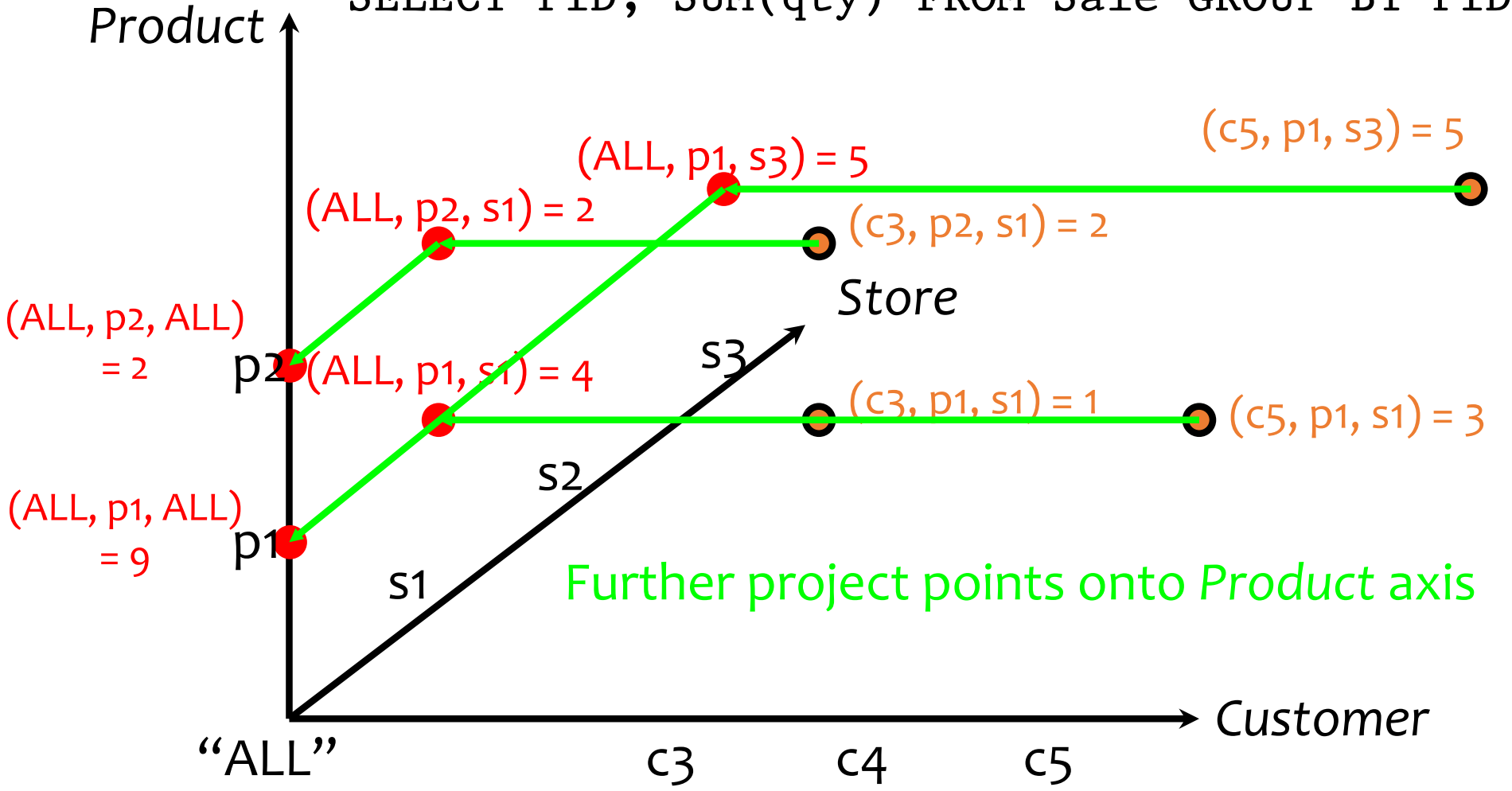
```
SELECT PID, SID, SUM(qty) FROM Sale
GROUP BY PID, SID;
```



# Completing the cube—axis

Total quantity of sales for each product

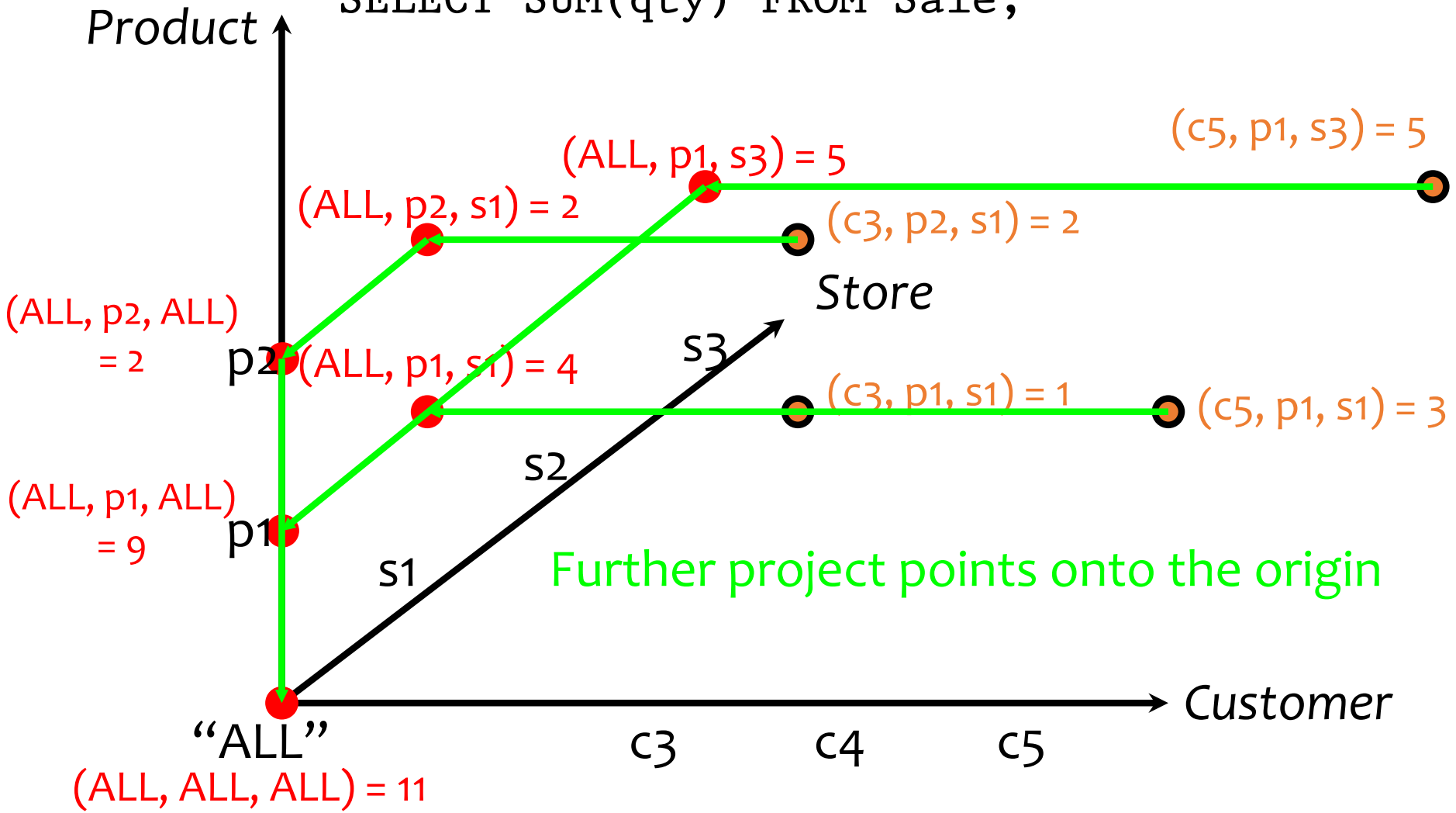
```
SELECT PID, SUM(qty) FROM Sale GROUP BY PID;
```



# Completing the cube—origin

Total quantity of sales

```
SELECT SUM(qty) FROM Sale;
```



# CUBE operator

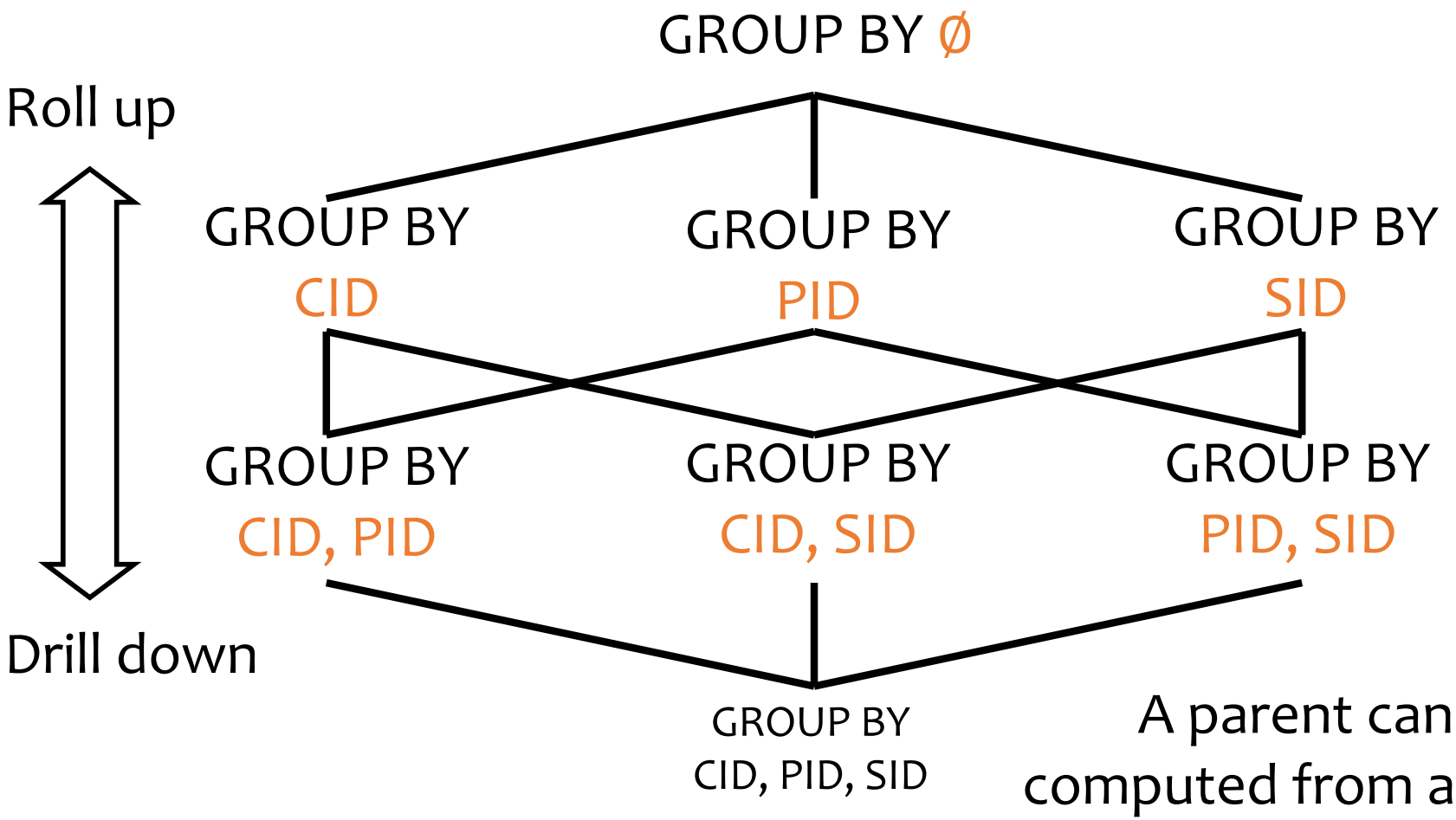
- *Sale* (*CID*, *PID*, *SID*, *qty*)
- Proposed SQL extension:

```
SELECT SUM(qty) FROM Sale
GROUP BY CUBE CID, PID, SID;
```

- Output contains:
  - 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.
- Can you write a CUBE query using only GROUP BY's?

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

# Aggregation lattice




A parent can be computed from any child

# Materialized views

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

☞ Idea: precompute and store the aggregates as materialized views

  - Maintained automatically as base data changes
  - No. 1 user-requested feature in PostgreSQL!

# 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 Ø is small, but not useful to most queries
  - GROUP BY CID, PID, SID is useful to any query, but too large to be beneficial

# Other OLAP extensions

- Besides extended grouping capabilities (e.g., `CUBE`), window operations have also been added to SQL
- A "window" specifies an ordered list of rows related to the "current row"
- A window function computes a value from this list and the "current row"
  - Standard aggregates: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`
  - New functions: `RANK`, `PERCENT_RANK`, `LAG`, `LEAD`, …

# RANK window function example

| sid | pid | cid | qty |
|---|---|---|---|
| Durham | beer | Alice | 10 |
| Durham | beer | Bob | 2 |
| Durham | chips | Bob | 3 |
| Durham | diaper | Alice | 5 |
| Raleigh | beer | Alice | 2 |
| Raleigh | diaper | Bob | 100 |

GROUP BY

| sid | pid | cid | qty |
|---|---|---|---|
| Durham | beer | Alice | 10 |
| | | Bob | 2 |
| Durham | chips | Bob | 3 |
| Durham | diaper | Alice | 5 |
| Raleigh | beer | Alice | 2 |
| Raleigh | diaper | Bob | 100 |

```
SELECT SID, PID, SUM(qty),
       RANK() OVER w
FROM Sale GROUP BY SID, PID
WINDOW w AS
(PARTITION BY SID
 ORDER BY SUM(qty) DESC);
```

Apply WINDOW after processing FROM, WHERE, GROUP BY, HAVING

- PARTITION defines the related set and ORDER BY orders it

E.g., for the following "row,"

| | | | |
|---|---|---|---|
| Durham | beer | Alice | 10 |
| | | Bob | 2 |

the related list is:

| | | | |
|---|---|---|---|
| Durham | beer | Alice | 10 |
| | | Bob | 2 |
| Durham | diaper | Alice | 5 |
| Durham | chips | Bob | 3 |

# RANK example (cont'd)

| sid | pid | cid | qty |
|---|---|---|---|
| Durham | beer | Alice | 10 |
| | | Bob | 2 |
| Durham | chips | Bob | 3 |
| Durham | diaper | Alice | 5 |
| Raleigh | beer | Alice | 2 |
| Raleigh | diaper | Bob | 100 |

```
SELECT SID, PID, SUM(qty),
       RANK() OVER w
FROM Sale GROUP BY SID, PID
WINDOW w AS
(PARTITION BY SID
 ORDER BY SUM(qty) DESC);
```

E.g., for the following "row,"

| | | | |
|---|---|---|---|
| Durham | beer | Alice | 10 |
| | | Bob | 2 |

the related list is:

| | | | |
|---|---|---|---|
| Durham | beer | Alice | 10 |
| | | Bob | 2 |
| Durham | diaper | Alice | 5 |
| Durham | chips | Bob | 3 |

Then, for each "row" and its related list, evaluate SELECT and return:

| sid | pid | sum | rank |
|---|---|---|---|
| Durham | beer | 12 | 1 |
| Durham | diaper | 5 | 2 |
| Durham | chips | 3 | 3 |
| Raleigh | diaper | 100 | 1 |
| Raleigh | beer | 2 | 2 |

# Multiple windows

| sid | pid | cid | qty |
|---|---|---|---|
| Durham | beer | Alice | 10 |
| | | Bob | 2 |
| Durham | chips | Bob | 3 |
| Durham | diaper | Alice | 5 |
| Raleigh | beer | Alice | 2 |
| Raleigh | diaper | Bob | 100 |

```
SELECT SID, PID, SUM(qty),
  RANK() OVER w,
  RANK() OVER w1 AS rank1
FROM Sale GROUP BY SID, PID
WINDOW w AS
  (PARTITION BY SID
   ORDER BY SUM(qty) DESC),
w1 AS
  (ORDER BY SUM(qty) DESC)
ORDER BY SID, rank;
```

No `PARTITION` means all "rows" are related to the current one

So `rank1` is the "global" rank:

| sid | pid | sum | rank | rank1 |
|---|---|---|---|---|
| Durham | beer | 12 | 1 | 2 |
| Durham | diaper | 5 | 2 | 3 |
| Durham | chips | 3 | 3 | 4 |
| Raleigh | diaper | 100 | 1 | 1 |
| Raleigh | beer | 2 | 2 | 5 |

# Summary

- Eagerly integrate data from operational sources and store a redundant copy to support OLAP
- OLAP vs. OLTP: different workload → different degree of redundancy
- SQL extensions: grouping and windowing

# Data mining

- Data → knowledge
- DBMS meets AI and statistics
- Clustering, prediction (classification and regression), association analysis, outlier analysis, evolution analysis, etc.
  - Usually complex statistical "queries" that are difficult to answer → often specialized algorithms outside DBMS
- We will focus on frequent itemset mining

# Mining frequent itemsets

- Given: a large database of transactions, each containing a set of items
  - Example: market baskets
- Find all frequent itemsets
  - A set of items $X$ is frequent if no less than $s_{min}\%$ of all transactions contain $X$
  - Examples: {diaper, beer}, {scanner, color printer}

| *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 |
| … | … |

# First try

- A naïve algorithm
  - Keep a running count for each possible itemset
  - For each transaction $T$, and for each itemset $X$, if $T$ contains $X$ then increment the count for $X$
  - Return itemsets with large enough counts
- Problem: The number of itemsets is huge!
  - $2^n$, where $n$ is the number of items
- Think: How do we prune the search space?

# The Apriori property

- All subsets of a frequent itemset must also be frequent
  - Because any transaction that contains $X$ must also contains subsets of $X$

☞If we have already verified that $X$ is infrequent, there is no need to count $X$'s supersets because they must be infrequent too

# The Apriori algorithm

Multiple passes over the transactions

- Pass $k$ finds all frequent $k$-itemsets (i.e., itemsets of size $k$)
- 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 |

Transactions

$s_{min}\% = 20\%$

| *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 |

Transactions

$s_{min}\% = 20\%$

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

Frequent 1-itemsets

Scan and count

| 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 |

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

# 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 |

Transactions

$s_{min}\% = 20\%$

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 |

Transactions

$s_{min}\% = 20\%$

Generate candidates

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

Frequent 3-itemsets

| itemset | count |
|---|---|

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

# Summary

- Only covered frequent itemset counting
- Skipped many other techniques (clustering, classification, regression, etc.)
- Compared with statistics and machine learning: more focus on massive datasets and I/O-efficient algorithms