Data Warehousing

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

Summer 2019

Part IV

Modelling Your Data

Business Process Measurements

Want to store information about **business processes**.

→ Store **"business process measurement events"**

Example: Retail sales

→ Could store information like: date/time, product, store number, promotion, customer, clerk, sales dollars, sales units, …

→ Implies a level of detail, or **grain**.

Observe: These stored data have different flavors:

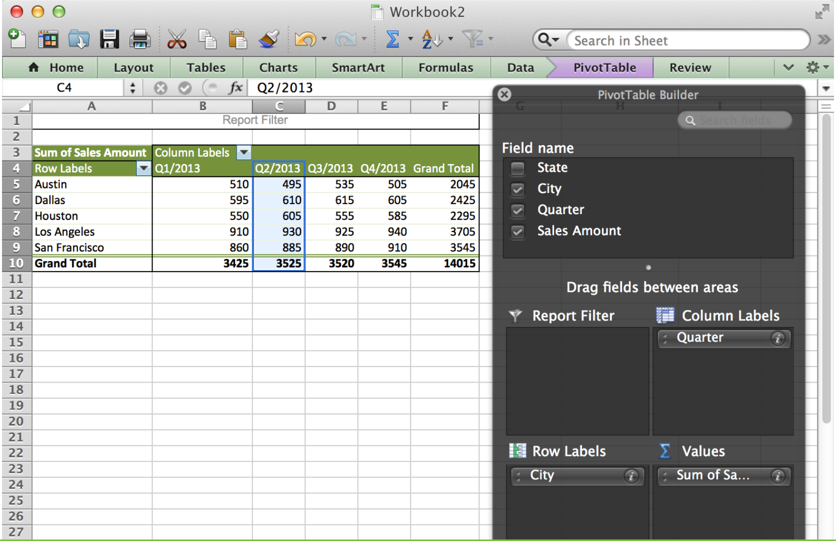
- Ones that refer to other entities, *e.g.*, to describe the context of the event (*e.g.*, product, store, clerk) (⇝ dimensions)
- Ones that look more like "measurement values" (sales dollars, sales units) (⇝ facts or measures)

Business Process Measurements Events

A flat table view of the events could look like

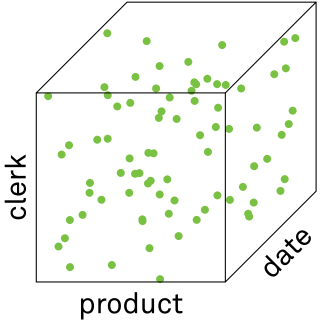
State	City	Quarter	Sales Amount
California	Los Angeles	Q1/2013	910
California	Los Angeles	Q2/2013	930
California	Los Angeles	Q3/2013	925
California	Los Angeles	Q4/2013	940
California	San Francisco	Q1/2013	860
California	San Francisco	Q2/2013	885
California	San Francisco	Q3/2013	890
California	San Francisco	Q4/2013	910
⋮	⋮	⋮	⋮

Business people are used to analyzing such data using **pivot tables** in **spreadsheet software**.



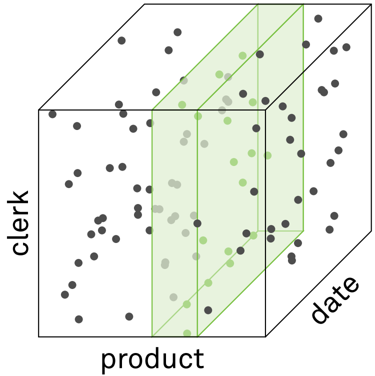
OLAP Cubes

Data cubes are alternative views on such data.



- Facts: points in the *k*-dimensional space
- **Aggregates** on sides and edges of the cube would make this a "*k*-dimensional pivot table".

More advanced analyses: "slice and dice" the cube.



- Specify range(s) along each dimension
- Aggregate over facts within these ranges.

→ **Dimensions** to define range

→ Aggregate **measures**

Advantage: Easy to understand

→ Users are analysts, not IT experts; want to do **ad hoc** analyses

Facts/Measures $\leftrightarrow$ Dimensions

Of the event table attributes, use some as **dimensions** and some as **measures** to aggregate.

Facts/measures:

- Fact: performance measure
- Typically **continuously valued**, almost always **numeric**
- They support sensible **aggregation**:
  - **additive facts:** Can be summed across any dimension
  - **semi-additive facts:** Can be summed across some, but not all dimensions
    E.g., account balance (can sum across customers, but not across dates)
  - **non-additive facts:** Cannot be meaningfully summarized
    E.g., item price, cost per unit, exchange rate

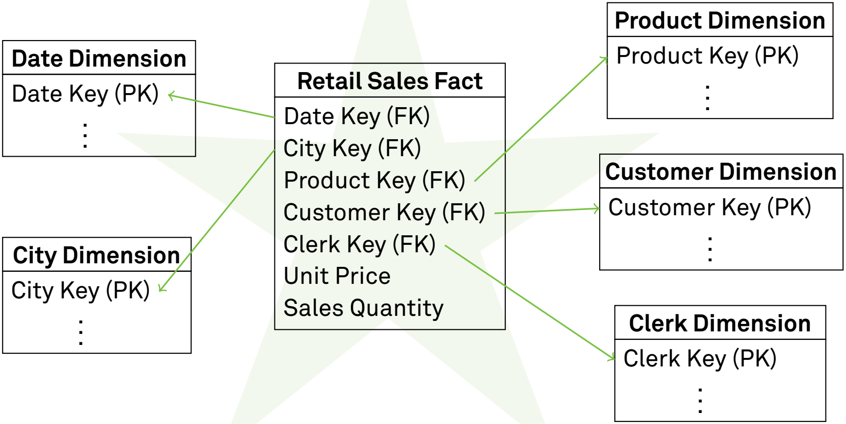
Facts/Measures $\leftrightarrow$ Dimensions

Dimensions:

- Typical criterion for **grouping**
- Many dimensions support some form of **hierarchy**
  - $\rightarrow$ *E.g.*, country $\rightarrow$ state $\rightarrow$ region $\rightarrow$ city
- Sometimes: more than one natural hierarchy
  - $\rightarrow$ *E.g.*, dates (year $\nearrow$ quarter $\longrightarrow$ month $\longrightarrow$ day ; year $\searrow$ week $\longrightarrow$ day)

Star Schema

Rather than a flat table, use a **star schema** for dimensional modelling in a **relational database**.



→ How will "slice and dice" queries look like on such a schema?

Four-Step Design Process

1. **Select the business process**

   E.g., start with a high impact, high feasibility business process; ↗ slide 38

2. **Declare the grain**

   Specify what exactly an individual fact table row represents.

   Examples:

   - One row per scan of a product in a sales transaction
   - One row per line item of a bill
   - One row per boading pass scanned at an airport gates
   - One row per daily snapshot of the inventory levels for each item in the warehouse

   If in doubt, use the smallest grain.

Four-Step Design Process

3. **Identify the dimensions**
   - Choose group-by criteria
   - The “who, what, where, when, why, and how” associated with the event.
   - Grain declaration $\leftrightarrow$ set of dimensions

4. **Identify the facts**
   - What is the process measuring?
   - Most (useful) facts are **additive**

Remember the **enterprise data warehouse bus matrix**?

	Date	Policy Holder	Coverage	Covered Item	Agent	Policy	Claim	Claimant	Payee
Underwriting Transactions	✔	✔	✔	✔	✔	✔			
Policy Premium Billing	✔	✔	✔	✔	✔	✔			
Agents' Commissions	✔	✔	✔	✔	✔	✔			
Claims Transactions	✔	✔	✔	✔	✔	✔	✔	✔	✔

→ **Rows:** business processes

→ **Columns:** dimensions

Example: Retail sales

- ✎ **Grain size?**
- ✎ **Dimensions?**
- ✎ **Facts?**

More on Dimensions—Product Dimension

E.g., **product dimension**

Possible attributes for product dimension table:

- Product Key (PK)
- Product Name
- Brand
- Category
- SKU Number (NK)
- Package Type
- Package Size
- Weight
- ...

Keys in dimension tables:

- Do **not** use operational keys ("natural keys", NK) to identify dimension tuples; use **surrogate keys** instead.
- May want to store natural key as additional dimension attribute.
- ✎ **Why?**

More on Dimensions—Date Dimension

If you're looking for dimensions, **date** is always a good guess.

Possible attributes:

- Date Key (PK)
- Day of Month
- Month Name
- Calendar Year
- Day of Week
- Week Number in Year
- Calendar Quarter
- ...

✎ **Huh?**

→ Why such a redundancy?

→ Why have a 'date' table at all?

Redundancy in Dimensions

Redundancy is **convenient.**

- *E.g.*, aggregate by week, without any date calculations
- Many functions on dates not supported by SQL
- Query **results** are more meaningful when they contain, *e.g.*, `'Monday'` rather than `1`.

Redundancy **won't hurt.**

- There at most 366 days per year
  - → Your date dimension table **will** remain small.
- Same argument holds true for most types of dimensions.
- No consistency problems as in transactional systems

Redundancy in Dimensions

In fact, redundancy is often used aggressively.

E.g., date dimension

- Fiscal Month
- Fiscal Year
- Fiscal Week
- Holiday Indicator
- Full Date as String
- SQL Date Stamp
- Calendar Year-Month
- ...

E.g., product dimension

- Category
- Sub Category
- Department Number
- Department Description
- Package Type
- Color
- ...

More on Dimensions—Flags and Indicators

Size of dimension tables is not usually a problem.

→ Store flags and indicators as **textual attributes**.

E.g.,

- `'Monday'`, `'Tuesday'`, … instead of `1`, `2`, …
- `'Non-Alcoholic'` and `'Alcoholic'` instead of `0` and `1`

✎ **Advantages?**

- Flags become self-explaining
  - → Did we start counting weekdays with `0` or `1`?
  - → Did `0`/`false` stand for 'alcoholic' or 'non-alcoholic'?

Normalizing / Snowflaking

Some designers feel they should **normalize**.

Product Dimension
Product Key (PK)
Product Name
Brand
Category (FK)
⋮

Category Dimension
Category Key (PK)
Category Name
Category Descr
Department (FK)
⋮

Department Dimension
Department Key (PK)
Department Number
Department Name
⋮

This is also referred to as **snowflaking**.

✎ **Consequences?**

→ Snowflaking is generally **not** a good idea.

→ More generally, normalization (as in the “Information Systems” course) is **not** a goal in DW modelling.

Remember the idea of pivot tables?

Sum of Sales Amount	Column Labels				
Row Labels	Q1/2013	Q2/2013	Q3/2013	Q4/2013	Grand Total
Austin	510	495	535	505	2045
Dallas	595	610	615	605	2425
Houston	550	605	555	585	2295
Los Angeles	910	930	925	940	3705
San Francisco	860	885	890	910	3545
Grand Total	**3425**	**3525**	**3520**	**3545**	**14015**

✎ **How can we express such functionality using SQL?**

OLAP Cubes and SQL—Dicing and Aggregation

Start situation: **flat table**

```
SELECT SUM (sales.quantity)
  FROM sales_flat AS sales
 WHERE sales.state = 'California'
   AND QUARTER (sales.date) = 3
```

With a **star schema**:

```
SELECT SUM (sales.quantity)
  FROM sales_fact AS sales, date_dimension AS d,
       store_dimension AS store
 WHERE sales.date_key = d.date_key
   AND sales.store_key = store.store_key
   AND store.state = 'California'
   AND d.quarter_of_cal_year = 3
```

OLAP Cubes and SQL—Grouping

Can also **group** by one or more criteria:

```
  SELECT store.state, d.quarter_of_cal_year,
         SUM (sales.quantity)
    FROM sales_fact AS sales, date_dimension AS d,
         store_dimension AS store
   WHERE sales.date_key = d.date_key
     AND sales.store_key = store.store_key
   GROUP BY store.state, d.quarter_of_cal_year
```

✎ **Can we build a pivot table from that?**

Modern SQL dialects offer functionality to group at **multiple** criteria at the same time.

```
SELECT store.state, d.quarter_of_cal_year, SUM(…)
  FROM sales_fact AS sales, date_dimension AS d, …
       ⋮
 GROUP BY CUBE(store.state, d.quarter_of_cal_year)
```

Effect:

```
STORE_CITY        QUARTER_OF_CAL_YEAR SUM_QTY
----------------- ------------------- ---------
-                                   3    192159
-                                   4    287972
-                                   -   1051150
Austin                              -    208001
Houston                             -    210481
Austin                              3     38542
Austin                              4     56734
Houston                             3     38385
```

OLAP Cubes and SQL—CUBE/ROLLUP

`CUBE(a, b, c)`:

- Group by all subsets of $\{a, b, c\}$

  $\rightarrow$ `(), (a), (b), (c), (a, b), (a, c), (b, c), (a, b, c)`

`ROLLUP(a, b, c)`:

- Group by all prefixes of $\{a, b, c\}$

  $\rightarrow$ `(), (a), (a, b), (a, b, c)`

`GROUPING SETS(…)`:

- Explicitly list all desired grouping sets, *e.g.*,

```
GROUP BY GROUPING SETS ((a, b),
                        (b, c),
                        (a, b, c))
```

Can also **combine** them, *e.g.*, `GROUP BY CUBE(a, b), ROLLUP(c, d)`

OLAP Cubes and SQL—CUBE/ROLLUP

Data analysis is an **explorative task.**

Typical scenario:

1. Make observation (*e.g.*, an exceptionally high/low value)
2. Investigate deeper ("Which city was responsible for the sales increase in that state?")
   → **Refine** grouping used before.
3. Repeat

The operation in Step 2 is also called **drill down**. The opposite operation (from fine to coarser grain) is called **rollup**.

→ CUBE/ROLLUP readily contain the information needed for **drill down**/**rollup**.

Null Values

Example:

- Weight stored as measure within a sales fact table.
  - → Some events may not have an associated weight.

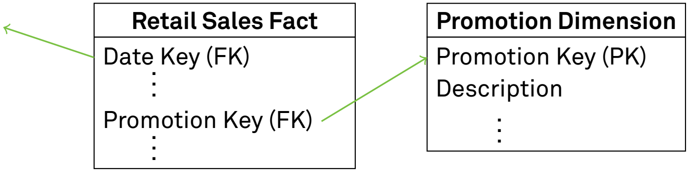
✎ How can we represent such absent measures?

- Store the **value/number 0**?

- Use a **null value**?

Null Values

Example: Information about promotions realized as a dimension



What about sales where we don't have an associated promotion?

- Null value in 'Promotion Key (FK)'? **No!**
  → ✎ What would happen during a join with tuples where 'Promotion Key (FK)' carries a null value?
- Instead: Insert explicit tuple into 'Promotion Dimension', *e.g.* "Not Applicable".

Null Values

Sometimes, there are multiple flavors of “Not Applicable”.

E.g., originally you might not have tracked promotions in your data warehouse. Once you add the new dimension, you end up with

1. old data where you have **no information** about promotions,
2. new data, where you **know** the sale happened without any promotion.

→ If you **don’t** represent absent values as `NULL`, those cases can trivially be represented as “Unknown”, “No Promotion”, ... dimension tuples.

Role Playing Dimensions

Consider an ‘Order’ business process.

Dimensions:

- Product
- Customer
- Handling Agent
- Shipping Method
- Order Date
- Requested Shipping Date

Two ‘Date’ Dimensions

Both ‘Date’ dimensions have the same value domain.

→ Implement as just one dimension table?

→ Tools might get confused about this.

Trick:

→ Use same **physical** ‘Date’ table, but create multiple **logical views** (‘Order Date’ view; ‘Requested Shipping Date’ view; etc.)

Degenerate Dimensions

For some dimensions, there are no sensible attributes to store.

E.g., transaction number on your sales receipt

→ Not much information to store for each transaction (beyond what's already stored as fact entries)

→ Yet, the transaction number is **useful**

  - Which products are often bought together?

Thus:

- Store the plain transaction number in the fact table
- Like a dimension, but no information can be found behind reference.
- We call this a **degenerate dimension**

Dealing with Updates

We haven't yet talked about **updates**.

Fortunately, ...

- DW workloads are **read-mostly**; update performance not critical
- ETL is the **only** updating process
  - → Update complexity less of an issue

Unfortunately, ...

- Updates **still** have to be dealt with
- Data warehouses contain **historic data**
  - → May have to **keep track** of changes

Dealing with Updates—Type 0

Type 0: “Retain Original” or “Passive Method”

- Once loaded, some dimension attributes can never change
    - *e.g.*, ‘in stock since’, ‘hire date’, ‘original credit score’
- Such attributes may be labeled “original”

$\rightarrow$ Type 0 attributes are **static**.

Dealing with Updates—Type 1

Type 1: "Overwrite"

- Similar to a normalized schema, **overwrite** old attribute values.
- *E.g.*, move 'IntelliKidz' software from 'Education' to 'Strategy' department:

Product Dimension			
Prod Key	SKU	Description	Department
12345	ABC922-Z	IntelliKidz	Education

↓

Product Dimension			
Prod Key	SKU	Description	Department
12345	ABC922-Z	IntelliKidz	Strategy

→ No keys or fact table entries are modified.

Dealing with Updates—Type 1

✎ **Pros and cons of this strategy?**

- Type 1 is a good mechanism to implement **corrections** in existing data.
- If previous values are not needed, simplicity of Type 1 may be appealing.

Dealing with Updates—Type 2

Type 2: "Add New Row"

- Don't overwrite, but create a **new dimension row**

Product Dimension					
Prod Key	SKU	Description	Department	Since	Until
12345	ABC922-Z	IntelliKidz	Education	1/1/12	12/31/99

↓

Product Dimension					
Prod Key	SKU	Description	Department	Since	Until
12345	ABC922-Z	IntelliKidz	Education	1/1/12	2/28/13
63726	ABC922-Z	IntelliKidz	Strategy	3/1/13	12/31/99

→ Old fact entries (still) point to old values, new to new.

→ Use addl. columns to track changes explicitly.

Dealing with Updates—Type 2

Effective and expiration dates:

- Explicitly store date of attribute change[2]
- Possibly store additional information
  - → Is this dimension row current?
  - → What is the key of the current dimension row?
  - → ...
- May simplify **ETL task**, too

Surrogate keys:

- Observe that Type 2 updates can only work with **surrogate keys**!
  - → *E.g.*, 'SKU' is no longer key in the above example

→ Type 2 is generally a good choice

---

[2] Use '12/31/99' to avoid trouble with null values.

Dealing with Updates—Type 3

Type 3: "Add New Attribute"

- Store current/previous information as **attributes**

Product Dimension			
Prod Key	SKU	Description	Department
12345	ABC922-Z	IntelliKidz	Education

↓

Product Dimension				
Prod Key	SKU	Description	Department	Prior Dept.
12345	ABC922-Z	IntelliKidz	Strategy	Education

→ Typical usage scenario: **company reorganization**

→ Don't use for attributes that change unpredictably!

Dealing with Updates—Type 4

Type 4: "Add Mini-Dimension"

Let's think about **Type 2** again:

→ What if changes are more frequent?

E.g., demographics information associated with customers

- age band (21–25; 26–30; 31–35; ...)
- income level ($<$ € 20,000; € 20,000–€ 24,999; ...)
- purchase frequency ('low', 'medium', 'high')

Problem: Profile updates can blow up dimension table by factors

Dealing with Updates—Type 4

Trick: Move volatile information to **separate dimension**, *e.g.*,

Demographics "Mini" Dimension			
Demogr Key	Age Band	Income Level	Purchase Frequency
1	21–25	<€ 20,000	low
2	21–25	<€ 20,000	medium
3	21–25	<€ 20,000	high
4	21–25	€ 20,000–€ 24,999	low
5	21–25	€ 20,000–€ 24,999	medium
6	21–25	€ 20,000–€ 24,999	high
⋮	⋮	⋮	⋮

→ 'Customer Dimension' no longer grows with updates.

→ 'Demographics Dimension' stays small (even under updates).

Again: Query Patterns

Analysis task: Relate customer calls to number of items sold.

product description	units sold	calls received
Footronic 08-15	417	38
Star Gizmo 42	976	296

→ This analysis relates **two business processes** to one another.

✎ **Can this analysis be expressed using SQL?**

Drilling Across Fact Tables

Combining business processes in such a way is called **drill across.**

The join in Step 3 assumes that products used in both business processes can **successfully be compared** (and find matches).

→ We say that the product dimensions must be **conformed.**

Case 1: Use same dimension tables

- Remember the enterprise data warehouse bus matrix?
  → Create one dim. table per column, one fact table per row.
- Conformed dimension tables must hold union of all values referenced by any fact table

Drilling Across Fact Tables

Case 1 (typically) requires that **grain sizes** of fact tables match.

Case 2: Rollup conformed dimension with attribute subset

- Coarser grain usually means that only a **subset** of the attributes applies.
- Remaining columns **must still conform**
  - → Use **same column labels**
  - → Ensure **same spelling** of all attribute values

Case 3: Shrunken conformed dimension with row subset

- Not all dimension rows may be relevant to all business processes
- *E.g.*, copy only relevant subsets to each department

Fact Table Types

All examples discussed so far assumed a **transactional fact table**.

→ Record business events, such as selling, shipping, stocking an items.

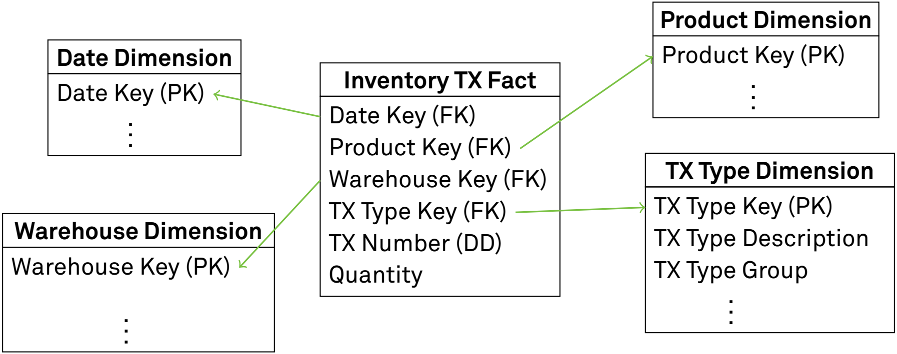
Suppose we want to keep an **inventory**.

→ Several transaction types will affect the inventory, *e.g.*,

- receive a product
- return product to vendor (because of a defect)
- place product in shelve
- pick product from shelve
- move product to a different shelve
- ship product to customer
- receive customer returns
- ...

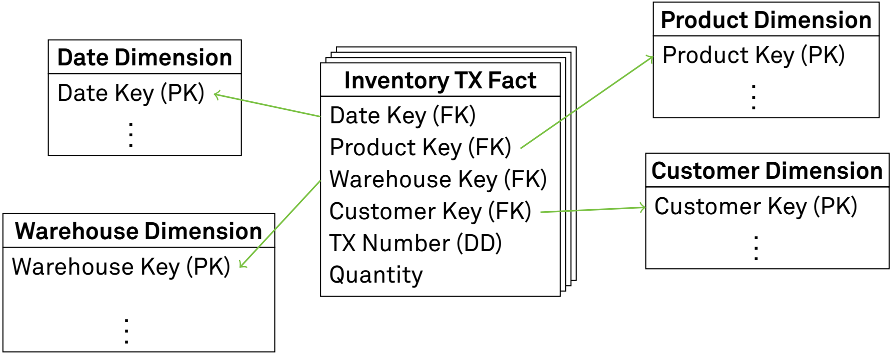
Modelling Inventory Transaction Types

Variant 1: Generic 'Inventory Transaction' fact table:



Modelling Inventory Transaction Types

Variant 2: One fact table per transaction type:



✎ **Pros/cons?**

Periodic Snapshots

For planning, **inventory levels** may be more relevant.

→ Transactions give us such informations only indirectly.

Instead: Periodic Snapshot Fact

Four-step dimensional design process:

1. **Business process: Periodic snapshotting** of inventory
2. **Grain:** daily, weekly, hourly, ... inventory levels
3. **Dimensions:** *e.g.*, date, warehouse, product
   → **not:** customer, promotion, ...
4. **Facts:** *e.g.*, quantity on hand

Semi-Additive Facts

Facts in periodic snapshot fact tables are usually **semi-additive**:

- Can aggregate across **some** dimensions.
  - → *e.g.*, total value of in-stock items
- But cannot aggregate across others, expecially date/time.
  - → *e.g.*, sum of inventory levels over one month makes no sense

Averages over snapshots make sense. But be careful to phrase queries correctly.

- Average over total warehouse value?

Accumulating Snapshot Fact Tables

Transaction fact table:

- Centered around **buying**/**selling**/**moving** stock items

Periodic snapshot fact table:

- Centered around **warehouse inventory level**.

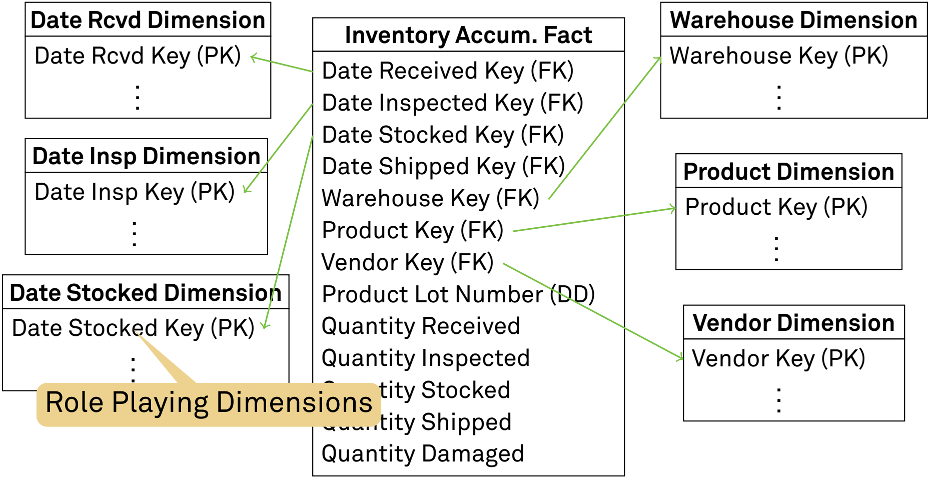
Accumulating snapshot fact table:

- Centered around **individual product item/lot**.

Idea:

- One fact table row per **product item/lot**.
- Store whereabouts of each item/lot as dimensions.

Inventory Accumulating Snapshot Fact Table



Inventory Accumulating Snapshot Fact Table

Update fact table as lot moves through value chain:

Inventory Accumulating Fact						
Date Rcvd	Date Insp	Date Stocked	···	Qty Rcvd	Qty Insp	···
20140214	0	0	···	42	–	···

↓

Inventory Accumulating Fact						
Date Rcvd	Date Insp	Date Stocked	···	Qty Rcvd	Qty Insp	···
20140214	20140215	0	···	42	40	···

↓

Inventory Accumulating Fact						
Date Rcvd	Date Insp	Date Stocked	···	Qty Rcvd	Qty Insp	···
20140214	20140215	20140215	···	42	40	···

Fact Table Types

We've seen three fact table types:

- transaction fact table
- periodic snapshot fact table
- accumulating snapshot fact table

All three are **complementary**.

→ Observe how they are designed around different processes.