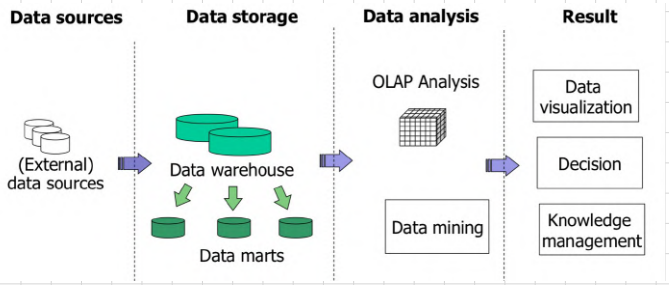
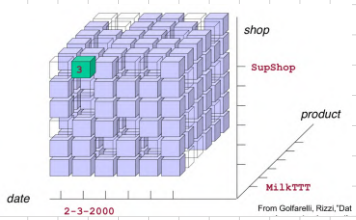
Summary

SUMMARY

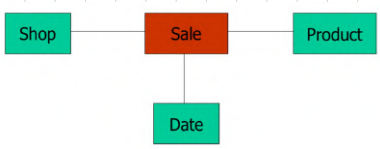
- **DATA WAREHOUSING:**
  - **INTRODUCTION:**
    - **DATA WAREHOUSING GOAL:**
      TRANSFORM DATA INTO USABLE INFORMATION FOR DIFFERENT PURPOSES



    - **TYPE OF INFORMATION PROCESSING:**
      - **TRASACTION PROCESSING:** ~> es. AMAZON SELLING PROCESS
        - OLTP: ON-LINE TRANSACTION PROCESSING
        -> TRADITIONAL DBMS USAGE
        - DETAILED DATA, SNAPSHOT OF CURRENT DATA, RELATIONAL REPRESENTATION
        - NEEDS TO BE FAST, FREQUENT UPDATES
      - **ANALYTICAL PROCESSING:** ~> es. AMAZON STORE DATA
        - OLAP: ON-LINE ANALYTICAL PROCESSING
        -> DECISION SUPPORT APPLICATION
        - HISTORICAL DATA, INTEGRATED AND CONSOLIDATED DATA
        - AD-HOC APPLICATION, SOMETIMES READ-ONLY
    - DATA WAREHOUSE IS KEPT FROM OPERATIONAL DB:
      -> THEY ARE USED FOR MAKING DECISIONS
      -> V: { PERFORMANCE -> COMPLEX QUERIES, DIFF. ACCESS METHODS AT PHYSICAL LEVEL
      DATA MANAGEMENT -> MISSING INFO, DATA CONSOLIDATION, DATA QUALITY
    - **DATA MODEL:**
      - HYPER-CUBE WITH 1 OR MORE DIMENSIONS



      - STAR MODEL FOR RELATIONAL REPRESENTATIONS

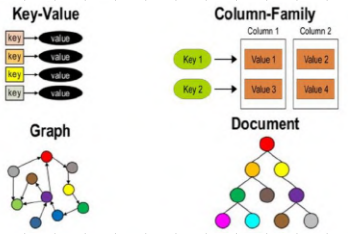


- SQL vs NOSQL DATA REPRESENTATION:

Relational database	NOSQL database
Table	Collection
Row	Document
Column	Field

  - DB → SET OF COLLECTIONS
    - COLLECTION → SET OF DOCUMENTS
      - DOCUMENT → LIST {KEY, VALUE} FIELDS

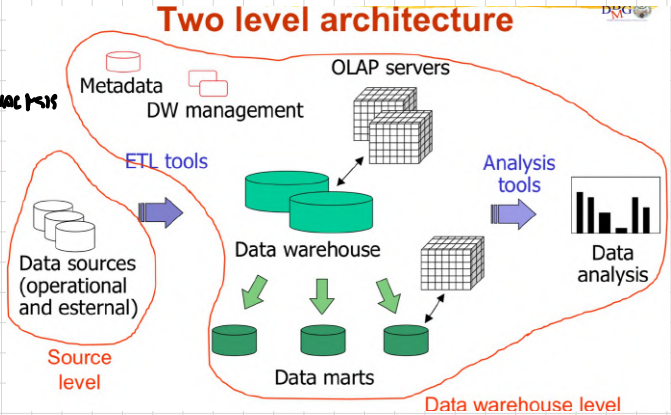
- DIFFERENT TYPES OF NOSQL DB:

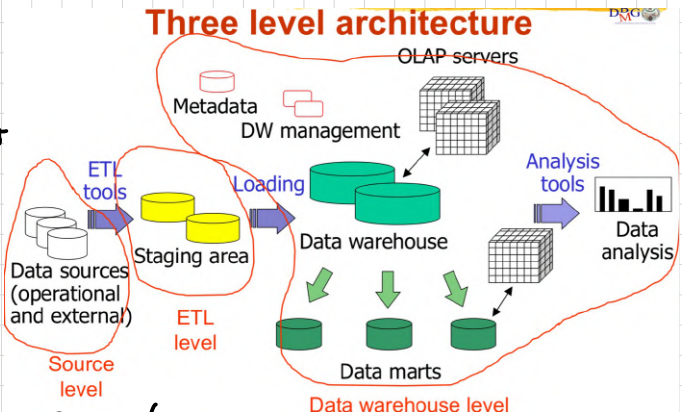


→ SIMILAR TO JSON

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
             phone: "123-456-7890",
             email: "xyz@example.com"
           },
  access: {
             level: 5,
             group: "dev"
           }
}
```

- DATA ANALYSIS:
  - KPI: KEY PERFORMANCE INDICATOR → MEASURABLE VALUES THAT CAN DEMONSTRATE COMPANY EFFECTIVENESS ON OBJECTIVE
- DATA VISUALIZATION: INFORMATIVE DASHBOARD
- DATA WAREHOUSE ARCHITECTURES:
  - SEPARATION BETWEEN COMPUTING AND DATA ANALYSIS
  - DATA MART: DEPARTMENTAL INFO SUBSET FOCUSED ON A GIVEN SUBJECT
  - SERVERS FOR DATA WAREHOUSE:
    - ROLAP (RELATIONAL OLAP):
      → EXTENDED RELATIONAL DBMS
    - MOLAP (MULTIDIM. OLAP):
      → DATA REPRESENTED IN MATRIX FORMAT
    - HOLAP (HYBRID): R + M OLAP
    - NO SQL ARCHITECTURES
    - ETL (EXTRACTION, TRANSFORMATION, LOADING):
      → PREPARES DATA TO BE LOADED INTO DW (EXTRACTION, TRANSFORMATION, CLEANING, LOADING)
    - METADATA: DATA ABOUT DATA





- 2 VS 3 LEVEL ARCHITECTURES:
  - 2 LEVEL: ETL "ON THE FLY" TRANSF. & CLEANING
  - 3 LEVEL: COMPLEX ETL → STAGING AREA

DATA WAREHOUSE DESIGN:

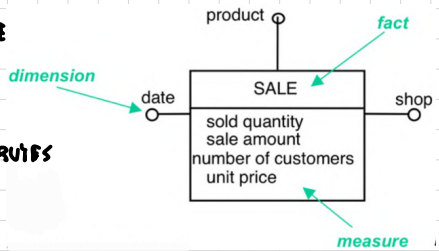
- FACTS: RELEVANT EVENTS FOR THE COMPANY

CONCEPTUAL DESIGN:

- THE ER MODELING IS NOT ADEQUATE
  → WE WILL USE THE DIMENSIONAL FACT MODEL (GOLFARELLI, RIZZI):

DIMENSIONAL FACT MODEL:

- FACT: SET OF RELEVANT EVENTS, IT EVOLVES WITH TIME
- DIMENSION: ANALYSIS COORDINATES OF A FACT, CHARACTERIZED BY MANY CATEGORICAL ATTRIBUTES
- MEASURE: NUMERICAL PROPERTY OF A FACT



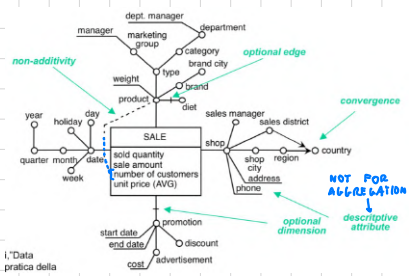
HIERARCHY:

- ATTRIBUTES DESCRIBE THE HIERARCHY AT DIFFERENT ABSTRACTION LEVELS
  → FUNCTIONAL DEPENDENCY:
  → 1:N RELATIONSHIP BETWEEN ATTRIBUTES

ex. ∀ SHOP → ∃! SHOP CITY
∀ SHOP CITY → ∩ SHOP

ADVANCED DFM ⇝

→ SAME LOGIC FOR EACH CONNECTION OF THE BRANCH:
└→ SHOP CITY/REGION, REGION/COUNTRY

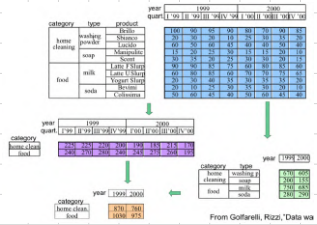


AGGREGATION:

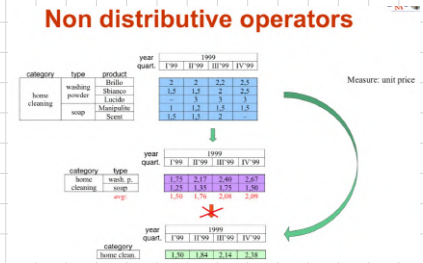
- IT COMPUTES MEASURES WITH A MAJOR GRANULARITY THAN THOSE IN THE ORIGINAL FACT SCHEMA
  → DETAIL REDUCTION: CLIMBING UP THE HIERARCHY
- MEASURE CHARACTERISTICS:
  - ADDITIVE
  - NOT ADDITIVE: CANNOT BE AGGREGATED ALONG A HIERARCHY WITH SUM OPERATOR
  - NOT AGGREGABLE

- **MEASURE CLASSIFICATION:**
  - **STREAM MEASURES:** CAN BE EVALUATED CONTINUOUSLY AT THE END OF A TIME PERIOD AGGREGATED BY MEANS OF ALL STANDARD OPERATORS
    ex. SOLD QUANTITY
  - **LEVEL MEASURES:** SNAPSHOT → EVALUATED AT A GIVEN TIME NOT ADDITIVE ALONG THE TIME DIMENSION
    ex. ACCOUNT BALANCE
  - **UNIT MEASURES:** EVALUATED AT A GIVEN TIME, EXPRESSED IN RELATIVE TERMS NOT ADDITIVE ALONG ANY DIMENSIONS
    ex. UNIT PRICE OF A PRODUCT

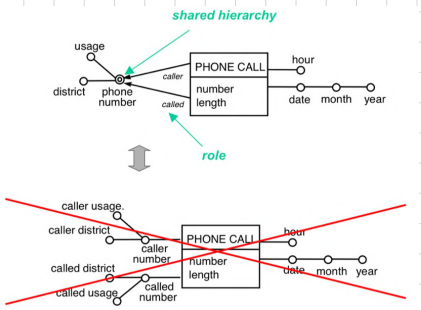
- **AGGREGATE OPERATORS:**



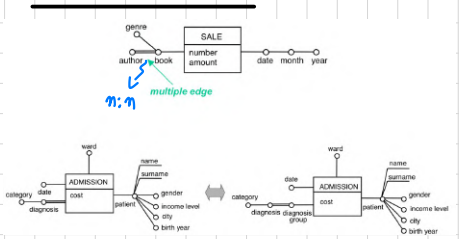
  - **DISTRIBUTIVE:**
    CAN ALWAYS COMPUTE HIGHER LEVEL AGGREGATIONS
    ex. SUM, MIN, MAX
  - **ALGEBRAIC:**
    CAN COMPUTE HIGHER LEVEL AGGREGATIONS ONLY WHEN MORE MEASURES ARE AVAILABLE
    ex. AVG (IT REQUIRES COUNT)
  - **OLISTIC:**
    CANNOT COMPUTE HIGHER LEVEL AGGREGATIONS
    ex. MODE, MEDIAN



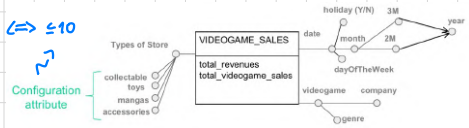
- **ADVANCED DFM:**
  - **SHARED HIERARCHY:**



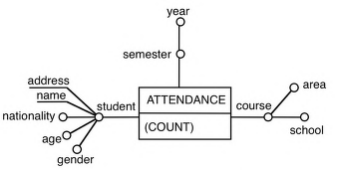
  - **MULTIPLE EDGE:**



  - **CONFIGURATION ATTRIBUTE:**



    - MORE VALUES AT THE SAME TIME
      → FACT ATTRIBUTE: BOOLEAN VALUE Y/N
  - **FACTLESS FACT SCHEMA:**



    - SOME EVENTS ARE NOT CHARACTERIZED BY MEASURES
      → IT RECORDS OCCURRENCE OF AN EVENT

- **REPRESENTING TIME:**
  - **TYPE 1:**

    DATA IS OVERWRITTEN WITH CURRENT VALUE

    -> USED WHEN THE DATA CHANGE REPRESENTATION IS NOT NEEDED

    ex. MARIO ROSSI CHANGE MARITIAL STATUS: SINGLE ↦ MARRIED

    ↳ ALL HIS PURCHASES CORRESPONDS TO "MARRIED" CUSTOMER

  - **TYPE 2:**

    EVENTS ARE RELATED TO THE TEMPORALLY CORRESPONDING DIMENSION VALUE

    -> AFTER EACH STATE CHANGE IN A DIMENSION, A NEW DIMENSION INSTANCE IS CREATED AND NEW EVENTS WILL BE RELATED TO THE NEW EVENTS

    ex. MARIO ROSSI: SINGLE ↦ MARRIED

    ↳ PURCHASES PARTITIONED IN THE "UNMARRIED" MARIO ROSSI AND "MARRIED" MARIO ROSSI (NEW INSTANCE) FOR NEXT PURCHASES

  - **TYPE 3:**

    ALL EVENTS ARE MAPPED TO A DIMENSION VALUE SAMPLED AT A GIVEN TIME

    -> EXPLICIT MANAGMENT OF DIMENSION CHANGE OVER TIME

    -> LIKE TYPE 2, BUT IT IS MODIFIED INTRODUCING:

    - TIMESTAMPS OF VALIDITY START/END
    - ATTRIBUTE FOR IDENTIFYING THE SEQUENCE OF MODIFICATIONS

    ex. MARIO ROSSI: SINGLE ↦ MARRIED

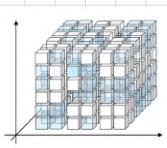
    -> SINGLE STATUS · VALIDITY END = TIMESTAMP OF MARRIAGE DATE

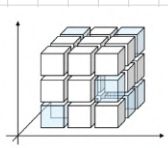
    -> MARRIED STATUS: VALIDITY START = TIMESTAMP OF MARRIAGE DATE

    -> NEW ATTRIBUTE TO TRACK ALL CHANGES TO MARIO ROSSI INSTANCE

- **WORK LOAD**
- **DATA VOLUME:**
  - **SPARSITY:** ↑ AGGREGATION LEVEL -> ↓ SPARCITY

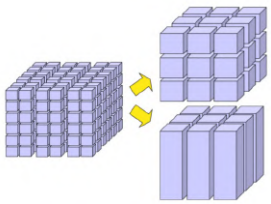
    -> IT AFFECT THE ACCURACY IN ESTIMATING AGGREGATED DATA CARDINALITY





DATA ANALYSIS, OLAP, EXTENDED SQL:

OLAP ANALYSIS:

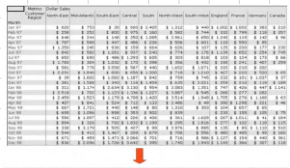


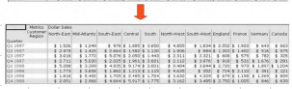
ROLL-UP:

DATA DETAIL REDUCTION BY:

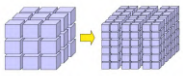
- DECREASING DETAIL DIMENSIONS, CLIMBING UP HIERARCHY
  -> AGGREGATE DATA AT A HIGHER LEVEL
  ex. GROUP BY STORE, MONTH ↦ GROUP BY CITY, MONTH
- DROPPING A WHOLE DIMENSION
  ex. GROUP BY PRODUCT, CITY ↦ GROUP BY PRODUCT

ex. MONTH ↦ QUARTER





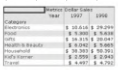
DRILL-DOWN: ~> OPPOSITE OF ROLL-UP

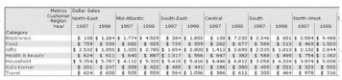


DATA DETAIL INCREASING BY:

- INCREASING DETAIL DIMENSIONS, GOING DOWN HIERARCHY
  ex. GROUP BY CITY, MONTH ↦ GROUP BY STORE, MONTH
- ADDING A WHOLE DIMENSION
  ex. GROUP BY PRODUCT ↦ GROUP BY PRODUCT, CITY

ex. < null > ↦ CUSTOMER REGION





SLICE AND DICE:

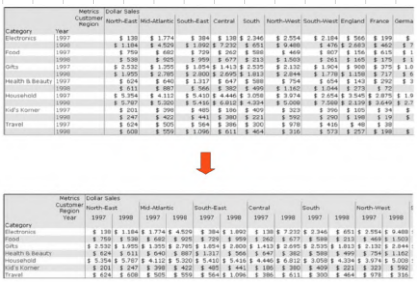
SELECTION OF A DATA SUBSET BY MEANS OF SELECTION PREDICATES

ex. YEARS = 2005, CATEGORY = "FOOD" AND CITY = "TORINO"

PIVOT:

RE-ORGANIZATION OF MULTI DIMENSIONAL STRUCTURE WITH NO VARIATION ON DETAIL LEVEL

-> INCREASE READABILITY OF THE SAME INFORMATION



- **EXTENDED SQL:**
  - NEW AGGREGATE FUNCTION: es. MOVING AVERAGE, MEDIAN
  - POSITION IN SORT ORDER
  - FUNCTION FOR REPORT GENERATION: RANKING AND CUMULATIVE TOTALS
  - DIFFERENT GROUP BY AT THE SAME TIME

EXAMPLE DB

City	Month	Amount
Milano	7	110
Milano	8	10
Milano	9	70
Milano	10	90
Milano	11	35
Milano	12	135
Torino	7	70
Torino	8	35
Torino	9	80
Torino	10	95
Torino	11	50
Torino	12	120

- **SQL OLAP FUNCTIONS:**
  - COMPUTATION WINDOW: COMPUTATION OF AGGREGATE FUNCTION IS PERFORMED INSIDE
    ↳ es. CUMULATIVE TOTALS, MOVING AVERAGE
  - NEW AGGREGATE FUNCTION TO COMPUTE RANK IN A GIVEN SORT ORDER
  - **COMPUTATION WINDOW:**
    CHARACTERIZED BY ('WINDOW' CLAUSE): DIFFERENT OF 'GROUP BY'
    - **PARTITIONING**: ROWS ARE GROUPED WITHOUT COLLAPSING THEM
    - **ROW ORDERING**: SEPARATELY IN EACH PARTITION (SIMILAR TO 'ORDER BY')
    - **AGGREGATION WINDOW**: ∀ ROW IN PARTITION, IT DEFINES THE ROW GROUP ON WHICH THE AGGREGATE FUNCTION IS COMPUTED

es.

- Show, for each city and month
  - sale amount
  - average on the current month and the two previous months, separately for each city
- Partitioning on city
  - average computation is reset when the city changes
- Ordering by month, to compute the moving average on the current month and the two preceding months
  - without ordering the computation is meaningless
- Aggregation window size: the current row and the two preceding rows

```
SELECT City, Month, Amount,
       AVG(Amount) OVER Wavg AS MovingAvg
FROM Sales
WINDOW Wavg AS (PARTITION BY City
                ORDER BY Month
                ROWS 2 PRECEDING)
```

NAME OF WINDOW · DECLARATION · INSTRUCTION · ∀ PARTITION · WINDOW: CURR ROW + 2 PREC.

OR

```
SELECT City, Month, Amount,
       AVG(Amount) OVER (PARTITION BY City
                         ORDER BY Month
                         ROWS 2 PRECEDING)
       AS MovingAvg
FROM Sales
```

City	Month	Amount	MovingAvg
Milano	7	110	110
Milano	8	10	60
Milano	9	90	70
Milano	10	80	60
Milano	11	40	60
Milano	12	140	90
Torino	7	70	70
Torino	8	30	50
Torino	9	80	60
Torino	10	100	70
Torino	11	50	60
Torino	12	150	100

- **N.B. ON es.:**
  - SORT ORDER IS REQUIRED TO COMPUTE AVG
  - WHEN WINDOW IS NOT COMPLETE (es. 1°, 2° ROW): COMPUTATION ON MADE ROWS
    → IT'S POSSIBLE TO REQUIRE A 'null' RESULT FOR EACH INCOMPLETE WINDOW
  - SEVERAL COMPUTATION WINDOW MAY BE SPECIFIED

- THE AGGREGATION WINDOW CAN BE DEFINED:
  - AT THE PHYSICAL LEVEL: IT BUILDS THE GROUP BY COUNTING ROWS
    - es. ROW 2 PRECEDING
    - es. ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    - es. ROWS UNBOUNDED PRECEDING/FOLLOWING
    - -> USEFUL IF THERE ARE NO "GAPS": es. MONTH: JAN, FEB, APRIL?
  - AT THE LOGICAL LEVEL: DISTANCE ON THE SORT KEY BETWEEN THE UPPER/LOWER BOUNDS ('RANGE' KEYWORD), CURR. VALUE SHOULD BE DEFINED
    - es. RANGE 2 MONTH PRECEDING, RANGE BETWEEN '<n>' DAY PRECEDING AND CURRENT ROW
    - -> USEFUL IF THERE CAN BE GAPS IN THE VALUES, es. MONTH IS MISSING IN A SEQUENCE
- APPLICATIONS: MOVING AGGREGATE COMPUTATION, CUMULATIVE TOTALS COMPUTATION, COMPARISON BETWEEN DETAILED AND AGGREGATED DATA

es. CUMULATIVE TOTALS

- Show, for each city and month
  - sale amount
  - cumulative sale amount for increasing months, separately for each city
- Partition by city
  - the cumulative total is reset when the city changes
- Order by (ascending) month to compute the sum for increasing months
  - without sorting, the computation would be meaningless
- Size of the aggregation window
  - from the starting row of the partition to the current row

WINDOW DEFINITION

```
SELECT City, Month, Amount,
       SUM(Amount) OVER (PARTITION BY City
                         ORDER BY Month
                         ROWS UNBOUNDED PRECEDING)
       AS CumeTot
FROM Sales
```

City	Month	Amount	CumeTot
Milano	7	110	110
Milano	8	10	120
Milano	9	90	210
Milano	10	80	290
Milano	11	40	330
Milano	12	140	470
Torino	7	70	70
Torino	8	30	100
Torino	9	80	180
Torino	10	100	280
Torino	11	50	330
Torino	12	150	480

es. COMPARISON BETWEEN DETAILED AND AGGREGATED DATA

- Show, for each city and month
  - sale amount
  - total sale amount on the whole time period for the current city
- Partition by city
  - the total amount is reset when the city changes
- Sorting is not needed
  - the total amount is computed independently of the sort order of tuples
- The aggregation window is not needed
  - it is the whole partition

```
SELECT City, Month, Amount,
       SUM(Amount) OVER (PARTITION BY City)
       AS TotalAmount
FROM Sales
```

City	Month	Amount	TotalAmount
Milano	7	110	470
Milano	8	10	470
Milano	9	90	470
Milano	10	80	470
Milano	11	40	470
Milano	12	140	470
Torino	7	70	480
Torino	8	30	480
Torino	9	80	480
Torino	10	100	480
Torino	11	50	480
Torino	12	150	480

es. COMPARISON BETWEEN DETAILED AND TOTAL DATA

- Show, for each city and month
  - sale amount
  - ratio between current row amount and grand total
  - ratio between current row amount and total amount by city
  - ratio between current row amount and total amount by month
- Three different computation windows
  - grand total: no partitioning
  - total by city: partition by city
  - total by month: partition by month
- No sort is needed in any window
  - totals are independent of the sort order of tuples
- The aggregation window is always the whole partition

```
SELECT City, Month, Amount
  Amount/SUM(Amount) OVER ()
  AS TotalFract
  Amount/SUM(Amount) OVER (PARTITION BY City)
  AS CityFract
  Amount/SUM(Amount) OVER (PARTITION BY Month)
  AS MonthFract
FROM Sales
```

City	Month	Amount	TotalFract	CityFract	MonthFrct
Milano	7	110	110/950	110/470	110/180
Milano	8	10	10/950	10/470	10/40
Milano	9	90	90/950	90/470	90/170
Milano	10	80	80/950	80/470	80/180
Milano	11	40	40/950	40/470	40/90
Milano	12	140	140/950	140/470	140/290
Torino	7	70	70/950	70/480	70/180
Torino	8	30	30/950	30/480	30/40
Torino	9	80	80/950	80/480	80/170
Torino	10	100	100/950	100/480	100/180
Torino	11	50	50/950	50/480	50/90
Torino	12	150	150/950	150/480	150/290

- **GROUP BY AND WINDOW:**

THEY CAN BE USED TOGETHER, WITH THE EXEC. ORDER:

-> GROUP BY -> WINDOW : WINDOW APPLY ON THE TABLE GENERATED FROM GROUP BY

es. GROUP BY AND WINDOW

- Assume that the **Sales** table contains information on sales with daily granularity
- Show, for each city and month
  - sale amount
  - average sale with respect to the current month and the two preceding months, separately for each city
- Grouping by month is needed to compute the total amount by month before computing the moving average
  - the group by clause is used for computing the monthly total
- The temporary table generated by the group by computation is the operand on which the computation window is defined

```
SELECT City, Month, SUM(Amount) AS TotMonth,
       AVG(SUM(Amount)) OVER (PARTITION BY City
                              ORDER BY Month
                              ROWS 2 PRECEDING)
       AS MovingAvg
FROM Sales
WHERE <join conditions>
GROUP BY City, Month
```

-> CANNOT USE "TOTMONTH" -> SQL PARSING PROBLEM

-> LINE 2: TOTMONTH NOT DEFINED YET

-> AVG OPERATES ON THE PARTITION, NOT SUM

- **RANKING FUNCTIONS:**
  - rank() : COMPUTES THE RANK, LEAVING EMPTY SLOT AFTER A TIE
    es. A : RANK 1, B: RANK 1, C : RANK 3
  - denseRank() : COMPUTES RANK, DOESN'T LEAVE EMPTY SLOT AFTER A TIE
    es. A : RANK 1, B: RANK 1, C : RANK 2

es. RANKING

- Show, for each city in december
  - sale amount
  - rank on amount
- Partitioning is not needed
  - a single partition including all cities
- Order by amount to perform ranking
  - without sorting, the computation would be meaningless
- The aggregation window is the whole partition

```
SELECT City, Amount,
       RANK() OVER (ORDER BY Amount DESC)
       AS Ranking
FROM Sales
WHERE Month = 12
```

City	Amount	Ranking
Torino	150	1
Milano	140	2

- **GROUP BY CLAUSE EXTENSION:**

- USUALLY MANY PARTIAL TOTALS ARE COMPUTED "IN ONE SHOT" -> AVOID MULTIPLE READING, REDUNDANT DATA SORTS

-> SQL-99 EXTENDED GROUP BY():

- ROLL UP
- CUBE
- GROUPING SETS

• ROLLUP :

COMPUTES AGGREGATIONS ON ALL GROUPS, BY REMOVING ONE BY ONE THE COLUMNS IN THE GROUPING CLAUSE

es.

- Consider the following tables
  - `Time(Tkey,Day,Month,Year,…)`
  - `Shop(Skey,City,Region,…)`
  - `Product(Pkey,PName,Brand,…)`
  - `Sales(Skey,Tkey,Pkey,Amount)`
- Compute total sales in the year 2000 for the following attribute combinations
  - product, month, city
  - month, city
  - city

```
SELECT City, Month, Pkey,
       SUM(Amount) AS TotSales
FROM Time T, Shop S, Sales V
WHERE T.Tkey = V.Tkey
  AND S.Skey = V.Skey
  AND Year = 2000
GROUP BY ROLLUP (City,Month,Pkey)
```

C, M, PK
C, M
C
C, M, PK
C, M
C
null

City	Month	Pkey	TotSales
Milano	7	145	110
Milano	7	150	10
Milano	…	…	…
Milano	7	NULL	8500
Milano	8	…	…
Milano	NULL	NULL	150000
Torino	…	…	150
Torino	…	NULL	2500
Torino	NULL	NULL	135000
…	…	…	…
NULL	NULL	NULL	25005000

• CUBE :

COMPUTES AGGREGATIONS ON ALL COMBINATIONS OF COLUMNS IN THE GROUPING CLAUSE

es.

- Compute total sales in the year 2000 for *all* combinations of the following attributes
  - product, month, city
- The following aggregations should be computed
  - product, month, city
  - product, month
  - month, city
  - product, city
  - product
  - month
  - city
  - no grouping

```
SELECT City, Month, Pkey,
       SUM(Amount) AS TotSales
FROM Time T, Shop S, Sales V
WHERE T.Tkey = V.Tkey
  AND S.Skey = V.Skey
  AND Year = 2000
GROUP BY CUBE (City,Month,Pkey)
```

• GROUPING SETS :

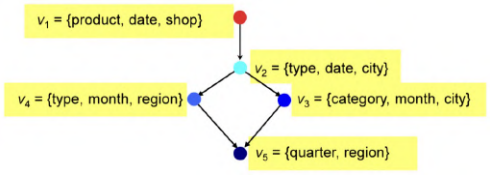
COMPUTES AGGREGATION ON THE GROUP LIST IN THE GROUPING CLAUSE

es.

- Compute total sales in the year 2000 for the following groups
  - month
  - month, city, product
- A roll up would perform the computation of unnecessary groupings and aggregations

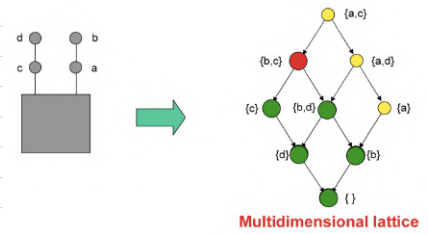
```
SELECT City, Month, Pkey,
       SUM(Amount) AS TotSales
FROM Time T, Shop S, Sales S
WHERE T.Tkey = S.Tkey
  AND S.Skey = S.Skey
  AND Year = 2000
GROUP BY GROUPING SETS
      (Month, (City,Month,Pkey))
```

- **MATERIALIZED VIEW**:

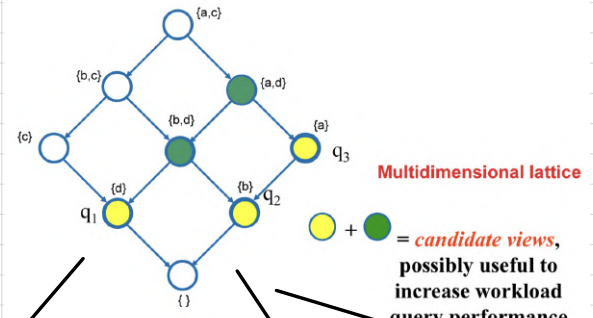


THEY ARE PRECOMPUTED SUMMARIES OF THE FACT TABLE

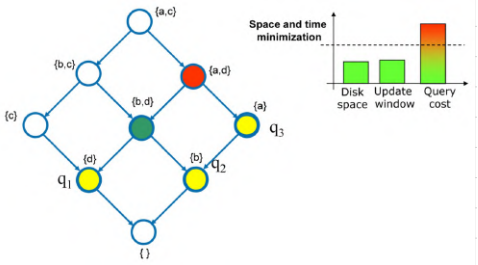
- EXPLICITLY STORED IN DW
- PERFORMANCE INCREASE FOR AGGREGATE QUERIES
- DEFINED STARTING FROM SQL STATEMENTS, LIKE GROUP BY

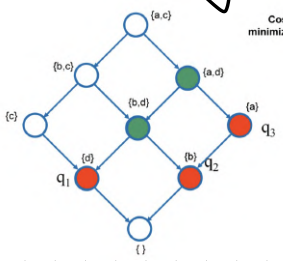


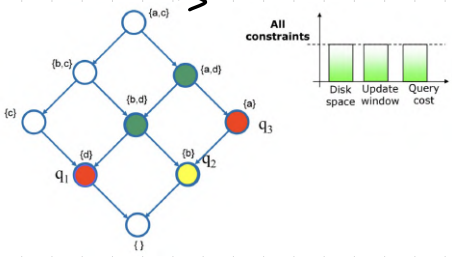
- **MATERIALIZED VIEW SELECTION**:
  - COST FUNCTION MINIMIZATION. QUERY EXECUTION COST, VIEW MAINTENANCE COST
  - CONSTRAINTS: AVAILABLE SPACE, TIME WINDOW FOR UPDATE, RESPONSE TIME, DATA FRESHNESS



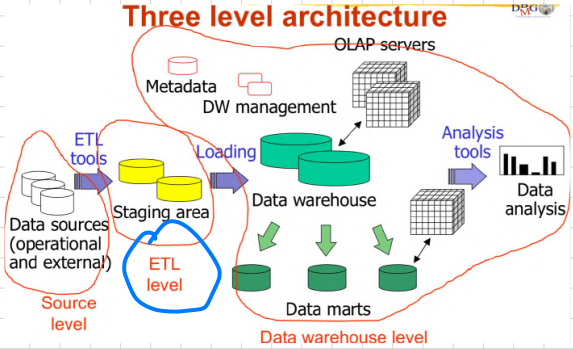
CONSIDERED MATERIALIZED VIEWS







- **ELT PROCESS** :



- ETL, EXTRACTION, TRANSFORMATION AND LOADING
- IT PREPARES DATA TO BE LOADED IN DW
- PERFORMED WHEN DW IS FIRST LOADED AND DURING PERIODICAL DW REFRESH

- **EXTRACTION** :

  - EXTRACTION METHODS :
    - STATIC : SNAPSHOT OF OPERATIONAL DATA, PERFORMED DURING 1ST DW POPULATION
    - INCREMENTAL : SELECTION OF UPDATES HAPPENED AFTER LAST EXTRACTION, PERFORMED FOR PERIODICAL DW REFRESH
      -> 2 TYPES : IMMEDIATE OR DEFERRED
  - IT DEPENDS ON HOW OPERATIONAL DATA IS COLLECTED :
    - HISTORICAL : ALL MODIFICATION STORED FOR A GIVEN TIME IN OLTP SYSTEMS
    - PARTLY HISTORICAL : ONLY A LIMITED N° OF STATES STORED IN OLTP SYSTEMS
    - TRANSIENT : ONLY CURRENT DATA STATE IN OLTP SYSTEM

  - **INCREMENTAL EXTRACTION** :
    - 4 TYPES :
      - APPLICATION ASSISTED : DATA MODIFICATIONS CAPTURED BY AD-HOC APPLICATION FUNCTIONS
      - LOG BASED : LOG DATA IS ACCESSED BY API's
      - TRIGGER BASED : TRIGGERS CAPTURE INTERESTING DATA MODIFICATION
      - TIMESTAMP BASED : MODIFIED RECORDS ARE MARKED WITH TIMESTAMP OF LAST MODIFICATION
    - **EXTRACTION TECHNIQUES** :

	Static	Timestamps	Application assisted	Trigger	Log
Management of transient or semi-periodic data	No	Incomplete	Complete	Complete	Complete
Support to file-based systems	Yes	Yes	Yes	No	Rare
Implementation technique	Tools	Tools or internal developments	Internal developments	Tools	Tools
Costs of enterprise specific development	None	Medium	High	None	None
Use with legacy systems	Yes	Difficult	Difficult	Difficult	Yes
Changes to applications	None	Likely	Likely	None	None
DBMS-dependent procedures	Limited	Limited	Variabile	High	Limited
Impact on operational system performance	None	None	Medium	Medium	None
Complexity of extraction procedures	Low	Low	High	Medium	Low

20.

4/4/2010

Cod	Product	Customer	Qty
1	Greco di tufo	Malavasi	50
2	Barolo	Maio	150
3	Barbera	Lumini	75
4	Sangiovese	Cappelli	45

6/4/2010

Cod	Product	Customer	Qty
1	Greco di tufo	Malavasi	50
2	Barolo	Maio	150
4	Sangiovese	Cappelli	145
5	Vermentino	Maltoni	25
6	Trebbiano	Maltoni	150

Incremental difference

Cod	Product	Customer	Qty	Action
3	Barbera	Lumini	75	D
4	Sangiovese	Cappelli	145	U
5	Vermentino	Maltoni	25	I
6	Trebbiano	Maltoni	150	I

- **DATA CLEANING:**

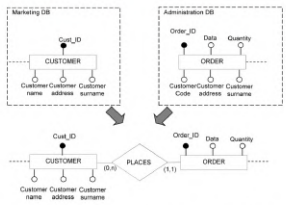
  - IMPROVING DATA QUALITY: DUPLICATES, MISSINGS, UNEXPECTED USES OF FIELDS, IMPOSSIBLE OR WRONG DATA VALUES, INCONSISTENCY

  -> ∀ PROBLEM -> AD-HOC TECHNIQUE, BETTER TO PREVENT THEM

  L> DATA DICTIONARY: ERROR -> SOLUTION

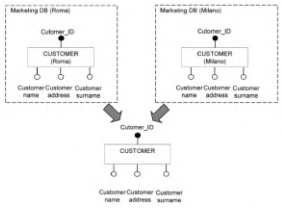
  L> APPROXIMATE FUSION: APPROXIMATE JOIN, PURGE/MERGE PROBLEM

Approximate join



- The join operation should be executed based on common fields, not representing the customer identifier

Purge/Merge problem



- Duplicate tuples should be identified and removed
- A criterion is needed to evaluate record similarity

Data cleaning and transformation example

Elena Baralis
C.so Duca degli Abruzzi 24
20129 Torino (I)

Normalization

name:	Elena
surname:	Baralis
address:	C.so Duca degli Abruzzi 24
ZIP:	20129
city:	Torino
country:	I

Standardization

name:	Elena
surname:	Baralis
address:	Corso Duca degli Abruzzi 24
ZIP:	20129
city:	Torino
country:	Italia

Correction

name:	Elena
surname:	Baralis
address:	Corso Duca degli Abruzzi 24
ZIP:	10129
city:	Torino
country:	Italia

- **TRASFORMATION:**

  CONVERSION: OPERATIONAL FORMAT ⟶ DW FORMAT

  -> UNIFORM DATA REPRESENTATION NEEDED

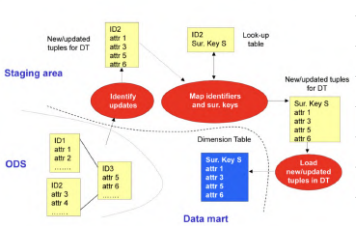
  - 2 STEPS: IN STAGING AREA
    - OPERATIONAL SOURCES ⟶ RECONCILED DATA:
      - CONVERSION AND NORMALIZATION, MATCHING, SIGNIFICANT DATA SELECTION
    - RECONCILED DATA ⟶ DW:
      - SURROGATE KEYS MANAGEMENT, AGGREGATE COMPUTATION

- **LOADING:**

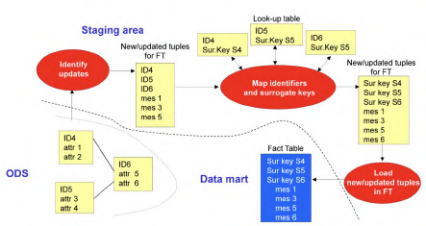
  UPDATE PROPAGATION TO DW, LIMITED TIME WINDOW TO PERFORM UPDATES, TRANSACTIONAL PROPERTIES NEEDED

  - UPDATE ORDER THAT PRESERVES DATA INTEGRITY: 1. DIMENSIONS, 2. FACT TABLES, 3. MATERIALIZED VIEWS AND INDEXES

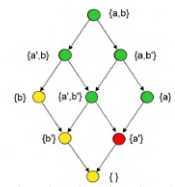
1. **Dimension table loading**



2. **Fact table loading**



3. **Materialized view loading**



- **PHYSICAL DESIGN :**

DESIGN OF THE PHYSICAL STRUCTURES FOR THE DB

- WORKLOAD CHARACTERIZATION
- PHYSICAL STRUCTURES
- OPTIMIZER CHARACTERISTICS : DATA ACCESS STRATEGIES
- PHYSICAL DESIGN PROCEDURE
- TUNING : VARIATION OF PHYSICAL STRUCTURES A POSTERIORI
- PARALLELISM

- **PHYSICAL ACCESS STRUCTURES :**

- PHYSICAL DATA STORAGE : SEQUENTIAL STRUCTURES, HASH STRUCTURES
- INDEXES TO INCREASE ACCESS EFFICIENCY : TREE STRUCTURES, UNCLUSTERED HASH INDEX, BITMAP INDEX

- **HEAP FILE :**
  - TUPLES SEQUENCED IN INSERTION ORDER : APPEND AT END OF FILE
  - DELETE/UPDATE MAY CAUSE WASTED SPACE
  - SEQUENTIAL READING/WRITING IS VERY EFFICIENT

- ORDERED SEQUENTIAL STRUCTURES :
  - USE OF SORT KEY
  - IF WE WANT TO PRESERVE THE ORDER -> LEAVE A % OF FREE SPACE ∀ BLOCK

- **B+ - TREE :** ~> B : BALANCED

DIRECT ACCESS TO DATA USING A KEY FIELD

- 2 TYPES :
  - UNCLUSTERED. POINTERS

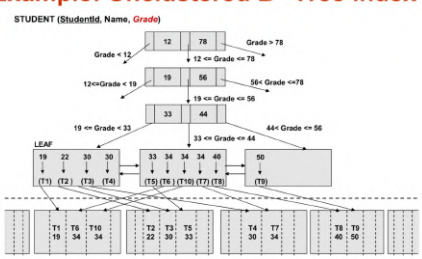
    LEAF CONTAINS PHYSICAL POINTERS TO ACTUAL DATA
  - CLUSTERED : DATA

    LEAF CONTAINS THE ACTUAL DATA

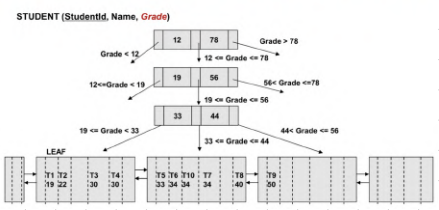
V : VERY EFFICIENT FOR RANGE QUERIES AND SCAN IN ORDER OF KEY FIELD

S : INSERTION AND DELETION

Example: Unclustered B+-Tree index



Example: Clustered B+-Tree index



- **BITMAP INDEX:**
  - MATRIX OF BITS 0/1: { ROWS: TUPLES ; COLUMNS: ATTRIBUTES }
  - DIRECT AND EFFICIENT ACCESS TO DATA, BY KEY FIELD
  - BASED ON BIT MATRIX:

    DATA ROWS IDENTIFIED BY RIDs

    { ROW: TUPLE ; COLUMN: DIFFERENT VALUE OF INDEXED ATTRIBUTE }

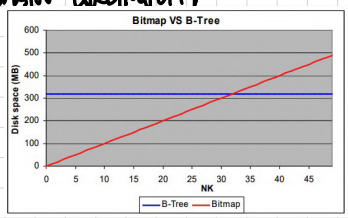
RID	Eng.	Cons.	Man.	Prog.	Assis.	Acc.
1	0	0	1	0	0	0
2	0	0	0	1	0	0
3	0	0	0	0	1	0
4	0	0	0	1	0	0
5	0	0	0	0	0	1

RID	Val_1	Val_2	...	Val_n
1	0	0	...	1
2	0	0	...	0
3	0	0	...	1
4	1	0	...	0
5	0	1	...	0

V: EFFICIENT FOR BOOLEAN EXPRESSION PREDICATES, APPROPRIATE FOR ATTRIBUTE WITH LIMITED DOMAIN

S: NOT USED FOR COMMONS ATTRIBUTES, REQUIRED SPACE & DOMAIN CARDINALITY

BITMAP <=> # KEYS < 30



- **JOIN INDEX:**

  IT PRECOMPUTES THE JOIN BETWEEN 2 TABLES

  -> IT STORES THE RID COUPLES OF TUPLES WHICH SATISFY THE PREDICATE

- **STAR-INDEX:**

  IT PRECOMPUTES THE JOIN RETURN MORE THAN 2 TABLES

- **BITMAPPED JOIN INDEX:**

  BIT MATRIX THAT PRECOMPUTES JOIN BETWEEN A DIMENSION AND A FACT TABLE

- **HASH STRUCTURE:**

  DIRECT AND EFFICIENT ACCESS TO DATA BASED ON A KEY FIELD, USING HASHING FUNCTION

  V: EFFICIENT FOR QUERIES WITH "=", NO SORTING REQUIRED

  S: COLLISIONS MAY OCCUR

  - UNCLUSTERED HASH INDEX: BLOCKS CONTAIN POINTERS TO DATA

- **INDEX CHOICE:**
  - INDEXING FOR DIMENSIONS { IN PREDICATE, ATTRIBUTE FREQ. INVOLVED ; DOMAIN HIGH CARDINALITY: B-TREE INDEX ; DOMAIN LOW CARDINALITY: BITMAP INDEX }
  - INDEXES FOR JOIN: BITMAPPED JOIN INDEXES RECOMMENDED
  - INDEXES FOR GROUP BY: USE MATERIALIZED VIEW

DATA WAREHOUSING IN ORACLE:

(EXAMPLES WITH ALREADY EXPLAINED CONCEPT SKIPPED)

ROW_NUMBER:

IN EACH PARTITION IT ASSIGNES A PROGRESSIVE NUMBER ∀ ROW

es.

Partition the items according to their type and enumerate in progressive order the data in each partition. In each partition the rows are sorted according to the weight

```
SELECT Type, Weight,
  ROW_NUMBER() OVER (
     PARTITION BY Type
     ORDER BY Weight
     ) AS RowNumberWeight
FROM Item;
```

Type	Weight	RowNumberWeight	
Bar	12	1	Partition 1
Gear	19	1	Partition 2
Screw	12	1	Partition 3
Screw	14	2	
Screw	16	3	
Screw	16	4	
Screw	16	5	
Screw	16	6	
Screw	17	7	
Screw	17	8	
Screw	18	9	
Screw	20	10	

CUME_DIST:

∀ PARTITION, IT ASSIGN A WEIGHT IN $[0;1]$ ACCORDINGLY TO THE $\%^{0}$ OF VALUES THAT PRECEDE IT, CONSIDERING THE SORTING IN THE PARTITION

es.

Partition the items according to the type and sort in each partition according to the weight of items. Assign to each row the corresponding value of CUME_DIST

```
SELECT Type, Weight,
  CUME_DIST() OVER (
     PARTITION BY Type
     ORDER BY Weight
     ) AS CumeWeight
FROM Item;
```

Type	Weight	RowNumberWeight		
Bar	12	1	(=1/1)	Partition 1
Gear	19	1	(=1/1)	Partition 2
Screw	12	0.1	(=1/10)	Partition 3
Screw	14	0.2	(=2/10)	
Screw	16	0.6	(=6/10)	
Screw	16	0.6	(=6/10)	
Screw	16	0.6	(=6/10)	
Screw	16	0.6	(=6/10)	
Screw	17	0.8	(=8/10)	
Screw	17	0.8	(=8/10)	
Screw	18	0.9	(=9/10)	
Screw	20	1	(=10/10)	

NTILE (n):

ALLOWS SPLITTING EACH PARTITION IN n IF POSSIBLE SUBGROUPS CONTAINING THE SAME AMOUNT OF ROWS

es.

Partition the items according to the type and split each partition in 3 sub-groups with the same number of data. In each partition the rows are ordered by the weight of items

```
SELECT Type, Weight,
  NTILE(3) OVER (
     PARTITION BY Type
     ORDER BY Weight
     ) AS Ntile3Weight
FROM ITEM;
```

Type	Weight	RowNumberWeight		
Bar	12	1		Partition 1
Gear	19	1		Partition 2
Screw	12	1		Partition 3
Screw	14	1	Subgroup 1	
Screw	16	1		
Screw	16	1		
Screw	16	2		
Screw	16	2	Subgroup 2	
Screw	17	2		
Screw	17	3		
Screw	18	3	Subgroup 3	
Screw	20	3		

- MATERIALIZED VIEW :

```
CREATE MATERIALIZED VIEW name
[BUILD IMMEDIATE/DEFERRED]
[REFESH (COMPLETE/FORCE/FAST/NEVER)(ON COMMIT/ON DEMAND)]
[ENABLE QUERY REWRITE]
AS query
```

- name: name of materialized view
- query: data from query to populate the materialized view
- BUILD:
  - IMMEDIATE: create view and immediately loads query result into the view
  - DEFERRED: not loaded at the moment
- REFRESH:
  - COMPLETE: recomputes value of views
  - FAST: updates the content of the materialized view using the changes since the last refresh. It needs the LOG materialized view
  - FORCE: when possible → FAST, otherwise COMPLETE
  - NEVER: views not updated using Oracle standard procedure
    - ON COMMIT: an automatic refresh is performed when SQL operations affect the materialized view content
    - ON DEMAND: the refresh is performed only upon explicit request of the user issuing the command `DBMS_MVIEW.REFRESH`
- ENABLE QUERY REWRITE:
  - enables DBMS to automatically use views as basic blocks to improve other queries performance

Ex. MATERIALIZED VIEW CREATION

```
CREATE MATERIALIZED VIEW Sup_Item_Sum
BUILD IMMEDIATE
REFRESH  COMPLETE ON DEMAND
ENABLE QUERY REWRITE
AS
   SELECT Cod_S, Cod_I, SUM(Measure)
   FROM Facts
   GROUP BY Cod_S, Cod_I;
```

- CHANGING MATERIALIZED VIEW : ALTER MATERIALIZED VIEW name, options;
- DELETE MATERIALIZED VIEW : DROP MATERIALIZED VIEW name

- COMMAND `DBMS_MVIEW.EXPLAIN_MVIEW` ALLOWS MATERIALIZED VIEW INSPECTION
  L> REFRESH TYPE, OP. ON WHICH FAST REFRESH IS ENABLED, QUERY REWRITE STATUS, ERRORS

NO SQL:

INTRODUCTION TO MONGO DB:

- RECORDS STORED IN BSON FORMAT (BINARY JSON)
- "_id": RESERVED FOR USE AS A PRIMARY KEY, IT'S AUTO GENERATED
- DATABASES AND COLLECTIONS: ~> INSTANCE > DB > COLLECTION > DOCUMENT (eg. LOG, LOG)
  - A INSTANCE OF MONGO DB -> N DATABASES, A DATABASE -> N COLLECTIONS
    - A COLLECTION -> SET OF DOCUMENTS
  - SHOW LIST OF AVAILABLE DBs: `show databases` (ANALOG FOR COLLECTIONS)
  - SELECT DIFFERENT DB: `use <database-name>`
  - DELETE/DROP DB: `db.dropDatabase()` ~> AFTER "USE <DB>"
  - CREATE COLLECTIONS: `db.createCollection(<collection name>, <options>);` ~> AFTER "USE <DB>"
  - DROP COLLECTION: `db.<collection_name>.drop()`
  - C.R.U.D OPERATIONS:

  - INSERT DOCUMENT IN A COLLECTION:

```
db.people.insertOne( {
        user_id: "abc123",
        age: 55,
        status: "A"
    } );
```

```
db.people.insertOne( {
        user_id: "abc124",
        age: 45,
        address: {
            street: "my street",
            city: "my city"
        }
} );
```

  - INSERT MULTIPLE DOCUMENTS:

```
db.people.insertMany([
 {user_id: "abc123", age: 55, status: "A"},
 {user_id: "abc124", age: 45, favorite_colors: ["blue", "green"]}
] );
```

  - DELETE:

```
db.people.deleteMany(
   { status: "D" }
)
```

MONGO DB, QUERYING DATA, FIND OPERATOR, AGGREGATION PIPELINES:

QUERYING DATA - find() OPERATION:

- find() IS THE ANALOG OF SELECT IN SQL

```
db.<collection name>.find( {<conditions>}, {<fields of interest>} );
```

SELECT id, user_id, status FROM people	db.people.find({ }, { user_id: 1, status: 1 })

Where Condition

Select fields

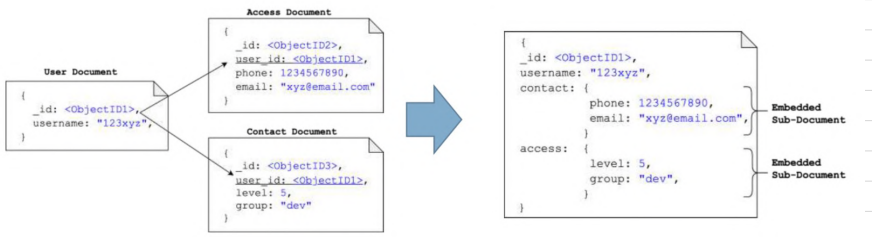
- SELECT A SINGLE DOCUMENT:

```
db.<collection name>.findOne( {<conditions>}, {<fields of interest>} );
```

- RELATION BETWEEN DOCUMENTS:

GIVEN BY "_id" AND DBref: `{ "$ref" : <value>, "$id" : <value>, "$db" : <value> }`

es.



- COMPARISON QUERY OPERATORS:

IN TUPLE <=> 1 OF a,b,c ↑ [a,b,c] ←

Name	Description
$eq or :	Matches values that are equal to a specified value
$gt	Matches values that are greater than a specified value
$gte	Matches values that are greater than or equal to a specified value
$in	Matches any of the values specified in an array
$lt	Matches values that are less than a specified value
$lte	Matches values that are less than or equal to a specified value
$ne	Matches all values that are not equal to a specified value, **including documents that do not contain the field.**
$nin	Matches none of the values specified in an array

es.

```
db.people.find(
    { age: { $gt: 25 } }
)
```

- NESTED: es "loc.type": "POINT"

- CONDITIONAL OPERATORS:

MySQL	MongoDB	Description
AND	,	Both verified
OR	$or	At least one verified

$OR: [{...}, {...}, ...]

es. AND

```
db.people.find(
    { status: "A",
      age: 50 }
)
```

es. OR

```
db.people.find(
{ $or:
  [ { status: "A" } ,
    { age: 50 }
  ]
}
)
```

(NO MORE SYNTAX -> SEE SLIDES INSTEAD)

- `db.inventory.find( { tags: ["red", "black"] } )` → Item list **order** matters!
  `db.inventory.find( { tags: { $all: ["red", "black"] } } )` → List order does **not** matter

  - Query an Array with **Compound Filter Conditions** on the Array Elements

- `db.inventory.find( { dim_cm: { $gt: 15, $lt: 20 } } )` → es. ooc: dim_cm: [30, 5] RETURNED (>15 ✓, <20 ✓)

  - Query for an Array Element that **Meets Multiple Criteria**

  Attention:
  - **Compound filter**: one element of the array can satisfy the greater than 15 condition and another element can satisfy the less than 20 condition, or alternatively a single element can satisfy both
  - **elemMatch**: one single element of the array **must** satisfy **both**

  `db.inventory.find( { dim_cm: { $elemMatch: { $gt: 15, $lt: 20 } } } )`

  ↳ MUST SATISFY BOTH CRITERIA

- **UPDATE** :

  `db.collection.updateOne(<filter>, <update>, <options>)`

  `db.collection.updateMany(<filter>, <update>, <options>)`

es.

```
db.inventory.updateMany(
   { "qty": { $lt: 50 } },
   {
     $set: { "size.uom": "in", status: "P" },
     $currentDate: { lastModified: true }
   }
)
```

es.

```
db.people.updateMany(
   { status: "A" } ,
   { $inc: { age: 3 } }
)
```

↳ AGE += 3

- **DATA AGGREGATION** :

es.

```
db.people.aggregate( [
   { $group: { _id: null,
               mytotal: { $sum: "$age" },
               mycount: { $sum: 1 }
             }
   }
] )
```

1 2 3

NAME

FIELD AS A VARIABLE

es.

```
db.people.aggregate( [
   { $match: {status: "A"} },
   { $group: { _id: null,
               count: { $sum: 1 }
             }
   }
] )
```

Where conditions

- **PIPELINE STAGES** :

Stage	Description
$addFields	Adds **new fields** to documents. Reshapes each document by adding new fields to output documents that will contain both the existing fields from the input documents and the newly added fields.
$bucket	Categorizes incoming documents **into groups**, called buckets, based on a specified expression and bucket boundaries. On the contrary, **$group** creates a "bucket" for each value of the group field.
$bucketAuto	Categorizes incoming documents into a specific number of groups, called buckets, based on a specified expression. Bucket boundaries are automatically determined in an attempt to **evenly distribute** the documents into the specified number of buckets.
$collStats	Returns statistics regarding a collection or view (it must be the **first stage**)
$count	Passes a document to the next stage that contains a count of the input **number of documents** to the stage (same as $group+$project)
$facet	Processes **multiple aggregation pipelines** within a single stage on the same set of input documents. Enables the creation of multi-faceted aggregations capable of characterizing data across multiple dimensions. Input documents are passed to the $facet stage only once, without needing multiple retrieval.
$geoNear	Returns an ordered stream of documents based on the **proximity** to a geospatial point. The output documents include an additional **distance** field. It must in the **first stage** only.
$graphLookup	Performs a **recursive search** on a collection. To each output document, adds a new array field that contains the traversal results of the recursive search for that document.
$group	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.
$indexStats	Returns statistics regarding the use of each index for the collection.
$limit	Passes the first n documents unmodified to the pipeline where n is the specified limit. For each input document, outputs either one document (for the first n documents) or zero documents (after the first n documents).
$lookup	Performs a **join** to another collection in the same database to filter in documents from the "joined" collection for processing. To each input document, the $lookup stage adds a new array field whose elements are the matching documents from the "joined" collection. The $lookup stage passes these reshaped documents to the next stage.
$match	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
$merge	Writes the resulting documents of the aggregation pipeline to a collection. The stage can incorporate (insert new documents, merge documents, replace documents, keep existing documents, fail the operation, process documents with a custom update pipeline) the results into an output collection. To use the $merge stage, it must be the last stage in the pipeline.
$out	Writes the resulting documents of the aggregation pipeline to a collection. To use the $out stage, it must be the last stage in the pipeline.
$project	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.
$sample	Randomly selects the specified number of documents from its input.
$set	Adds new fields to documents. Similar to $project, $set reshapes each document in the stream; specifically, by adding new fields to output documents that contain both the existing fields from the input documents and the newly added fields. $set is an alias for **$addFields** stage. If the name of the new field is the same as an existing field name (including _id), $set **overwrites** the existing value of that field with the value of the specified expression.
$skip	Skips the first n documents where n is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first n documents) or one document (if after the first n documents).
$sort	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.
$sortByCount	Groups incoming documents based on the value of a specified expression, then computes the count of documents in each distinct group.
$unset	Removes/excludes fields from documents.
$unwind	Deconstructs an array field from the input documents to output a document for each element. Each output document replaces the array with an element value. For each input document, outputs n documents where n is the number of array elements and can be zero for an empty array.

- **MONGO DB INDEXES:**

  - INDEXES ARE DATA STRUCTURES WHICH HELP MONGO DB TO TRAVERSE THE WHOLE COLLECTION
  - THEY STORE ORDERED VALUE OF SPECIFIC FIELDS
  - `db.collection.createIndex(<index keys>, <options>)`
    - INDEX KEYS : FIELD TO STORE AND -1 / 1 FOR SORT ORDER
    - OPTIONS : NAME, UNIQUE (NOT ACCEPT DOCUMENTS WITH DUPLICATE KEYS), etc.

  -> # FIELDS = 1 -> SINGLE INDEX, # FIELDS ≥ 2 -> COMPOUND INDEX

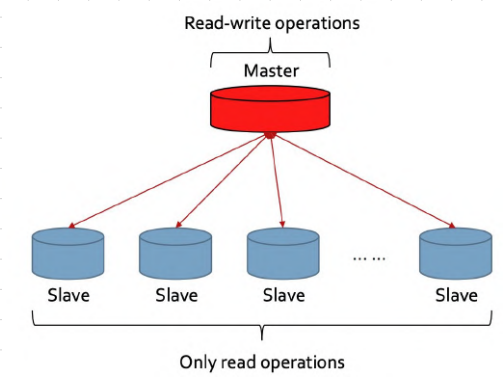
- **DISTRIBUTED DATA MANAGEMENT, REPLICATION, CAP THEOREM:**

- REPLICATION: SAME DATA IN DIFFERENT PLACES (OR PORTION OF SAME DATASET)

V: REDUNDANCY HELPS SURVIVING FAILURES (AVAILABILITY)
BETTER PERFORMANCES

-> HOW TO: MASTER-SLAVE REPLICATION, ASYNC REPLICATION

- **MASTER-SLAVE REPLICATION:**

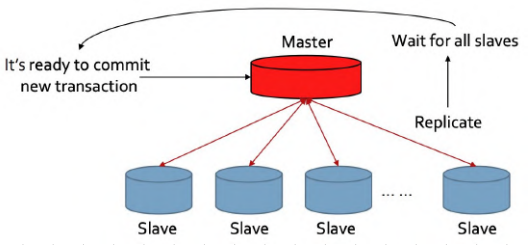


- ONLY READ SCALABILITY:

  - MASTER: ALL INSERT, WRITE, UPDATE
  - SLAVE: ALL READ

-> MASTER IS A SINGLE POINT OF FAILURE

- **SYNCRONOUS REPLICATION:**

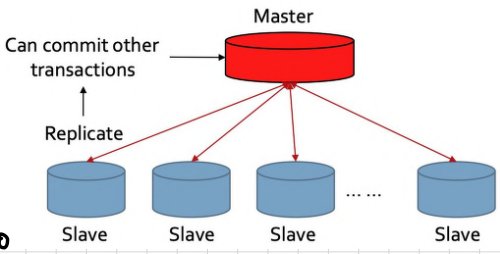


MASTER WAITS FOR ALL SLAVES TO COMMIT BEFORE COMMITTING A TRANSACTION

S: PERFORMANCE KILLER -> SLOWER

- BETTER APPROACH: WAIT FOR A SLAVE SUBSET

- **ASYNCRONOUS REPLICATION:**



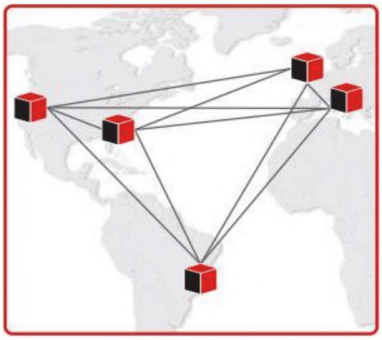
MASTER DO NOT WAIT FOR SLAVES, IT COMMITS LOCALLY

- EACH SLAVE FETCHES UPDATE INDEPENDENTLY
  - IF NO SLAVE HAS REPLICATED -> CHANGES LOST
  - IF SOME HAVE REPLICATED AND SOME NOT -> NEED TO RECONCILE

V: FASTER

S: UNRELIABLE

- **DISTRIBUTED DATABASES:**



DIFFERENT AUTONOMOUS MACHINES, WORKING TOGETHER TO MANAGE SAME DATASET

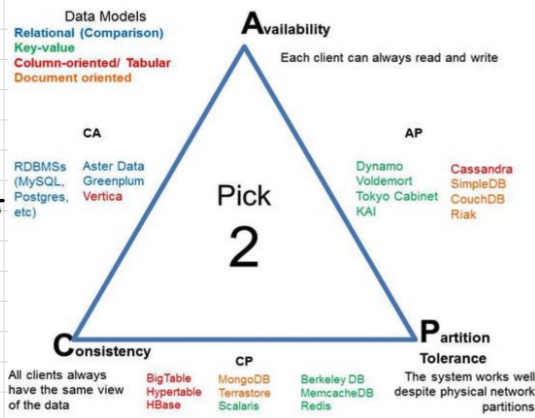
- 3 TYPICAL PROPERTIES IN DISTRIBUTED DB:
  - **CONSISTENCY**: ALL DISTRIBUTED DBs PROVIDES THE SAME DATA TO THE APPLICATION
  - **AVAILABILITY**: DB FAILURE DO NOT PREVENT SURVIVORS FROM CONTINUING TO OPERATE
  - **PARTITION TOLERANCE**: DB ABILITY TO WORK DESPITE PHYSICAL NETWORK PARTITIONS

-> PRACTICAL DB CAN PROVIDE ONLY 2/3 FEATURES ~> CAP THEOREM

- **CA WITHOUT P (LOCAL CONSISTENCY)**:
  - PARTITIONING CAN CAUSE A FAILURE

  -> IT PROVIDES LOCAL CA RATHER THAN GLOBAL CA

  -> MULTIPLE INDEPENDENT SYSTEMS WITH 100% CA



- **CP WITHOUT A (TRANSACTION LOCKING)**:
  - SYSTEM IS ALLOWED TO NOT ANSWER REQUESTS AT ALL

  -> IF A PARTITION/FAULTS OCCURS, ALL RESPONSES ARE BLOCKED UNTIL C IS AGAIN VERIFIED

  -> ACCESS BLOCK TO REPLICA SETS THAT AREN'T IN SYNC

- **AP WITHOUT C (BEST EFFORT)**:

  EVERY PARTITION MAKE AVIABLE WHAT IT KNOWS, DESPITE MAY HAVING DIFFERENT DATA

  -> NO ASSURANCE OF GLOBAL CONSISTENCY AT ALL TIMES

-> C, A, P ARE NOT BINARY PROPERTY, THEY ARE CONTINUOS FROM 0 TO 100%

-> USUALLY P ARE RARE, BUT IF ∃, A/C % NEEDS TO BE SACRIFIED

- **ACID PROPERTIES:**

F/S - **ATOMICITY:** DB TRANSACTION MUST FAIL / SUCCESS COMPLETLY; PARTIAL NOT ALLOWED

$V_i \rightarrow V_j$ - **CONSISTENCY:** TRANSACTION: VALID STATE ⟶ NEW VALID STATE

INDIP. - **ISOLATION:** ∀ TRANSACTION IS INDEPENDENT FROM OTHERS

t ⟶ - **DURABILITY:** RESULTS OF A TRANSACTION MUST BE PERMANENT

- **BASE PROPERTIES:** ~> LESS STRONG THEN ACID, (BASE)

cAP - **BASICALLY AVIABLE:** SYSTEM PROVIDES AVAILABILITY, IN TERMS OF CAP THEOREM

≈D - **SOFT STATE:** STATE OF A SYSTEM MAY CHANGE OVER TIME, EVEN WITH NO INPUT

t ⟷ C - **EVENTUAL CONSISTENCY:** SYSTEM WILL BECOME CONSISTENT OVER TIME

- **ACID VS BASE:**
  - ACID -> FOCUS ON CONSISTENCY, TRADITIONAL DB APPROACH ~> DATA ALWAYS CONSISTENT C
  - BASE -> FOCUS ON AVAILABILITY ~> DATA ALWAYS AVAILABLE A
    WORKS WELL IN PRESENCE OF PARTITION, PROMOTES AVAILABILITY

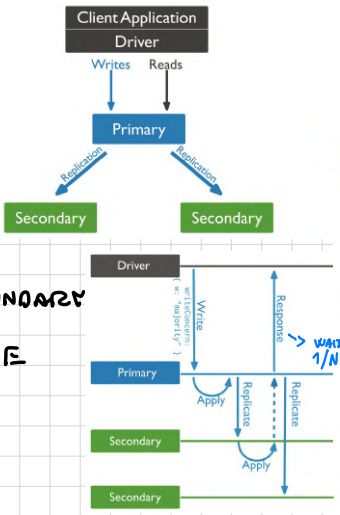
- **CONFLICT DETECTION AND RESOLUTION:**

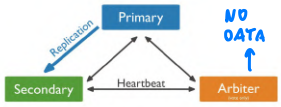
ex. A, B CUSTOMERS: A BOOKS LAST AVILABLE HOTEL ROOM
B DOES THE SAME ON A DIFFERENT NON-CONSISTENT NODE
-> 2 CONFLICTING UPDATES

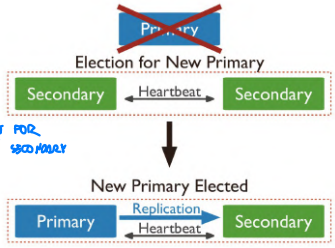
- APPLICATION SHOULD SOLVE THE CONFLICT WITH CUSTOM LOGIC
- DB CAN:
  - DETECT THE CONFLICT
  - PROVIDE LOCAL SOLUTION (ex. LATEST VERSION SAVED AS WINNING)

- COUCH DB: GUARANTEED THAT EACH INSTANCE THAT SEES SAME CONFLICT -> SAME SOLUTION
  ↳ IT RUNS A DETERMINISTIC ALGORITHM TO PICK WINNER
  -> REVISION WITH LONGEST REVISION LIST BECOMES WINNING REVISION

- **REPLICATION IN MONGO DB :**



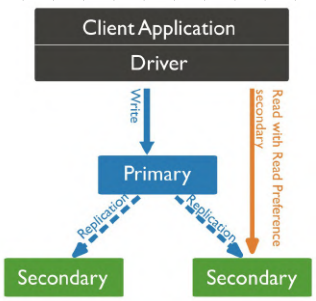




- FOR WRITE CONCERN OF W=1 OR W: "majority" :
  - -> PRIMARY WAITS UNTIL N SECONDARY ACKNOWLEDGE THE WRITE, BEFORE RETURNING WRITE CONCERN ACK
- AUTOMATIC FAILOVER (=> PRIMARY DOESN'T COMMUNICATE WITH OTHERS AFTER ϵ SECONDS
  - -> REPLICA CAN CONTINUE TO SERVE (=> BETTER IF # IS ODD (ELECTION OF PRIMARY)) QUERIES ARE CONFIGURED TO RUN ON SECONDARIES
  - MEDIAN TIME FOR PRIMARY ELECTION $\approx$ < 12 s

Number of Members	Majority Required to Elect a New Primary	Fault Tolerance
3	2	1
4	3	1
5	3	2
6	4	2

- **READ OPERATIONS :**
  - DEFAULT : CLIENT READS FROM PRIMARY ;
  - ASYNC REPLICATION :
    - READ FROM SECONDARIES MAY RETURN DATA THAT DO NOT REFLECT DATA ON PRIMARY



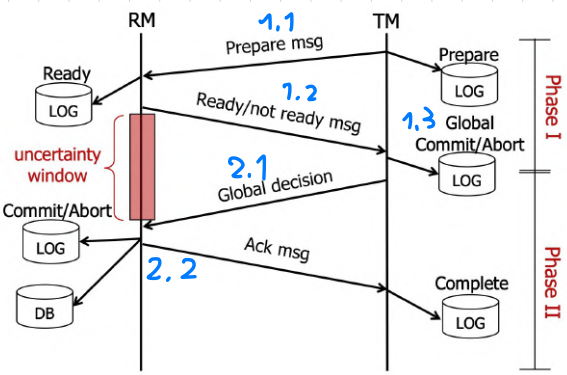
- ALL OPERATIONS IN A GIVEN TRANSACTION MUST ROUTE TO THE SAME MEMBER
  - -> UNTIL A TRANSACTION COMMITS, DATA CHANGES MADE IN THE TRANSACTION ARE NOT VISIBLE OUTSIDE TRANSACTION

(mongoDB EXPLANATION)

- **DISTRIBUTED TRANSACTIONS:**
  - **ACID PROPERTIES:**
    - A: REQUIRES DISTRIBUTED TECHNIQUES (2 PHASE COMMIT)
    - C: CONSTRAINTS CURRENTLY ENFORCED ONLY LOCALLY
    - I: IT REQUIRES STRICT 2 PHASE LOAD AND 2 PHASE COMMIT
    - D: REQUIRES EXTENSION OF LOCAL PROCEDURES TO MANAGE ATOMICITY IF ∃ FAILURE
  - **DISTRIBUTED QUERY OPTIMIZATION:**
    USER QUERY IS OPTIMIZED TO BE EXECUTED ON DISTRIBUTED NODES
  - ALL NODES INCLUDED IN A DISTRIBUTED TRANSACTION MUST IMPLEMENT SAME DECISIONS (COMMIT OR ROLLBACK) → COORDINATED BY 2 PHASE COMMIT
  - **2 PHASE COMMIT PROTOCOL:**

    GOAL: COORDINATION OF THE CONCLUSION OF A DISTRIBUTED TRANSACTION
    - 1 COORDINATOR: TRANSACTION MANAGER (TM)
    - N DBMS SERVERS FOR THE TRANSACTION: RESOURCE MANAGERS (RM)
    - ANY PARTICIPANT CAN BECOME A TM
    - LOG RECORDS:
      - TM LOG:
        - PREPARE: CONTAINS NODE ID + PROCESS ID OF ALL RMs PARTICIPANTS
        - GLOBAL COMMIT/ABORT: FINAL DECISION ON TRANSACTION OUTCOME
        - COMPLETE: WRITTEN AT PROTOCOL END
      - RM LOG:
        - READY: RM IS WILLING TO PERFORM COMMIT OF THE TRANSACTION
          → AFTER THAT, RM LOSES ITS AUTONOMY FOR CURRENT TRANSACTION

- **PROTOCOL:**



1.

1.1 **TM**: WRITE THE PREPARE RECORD IN LOG, SENDS THE PREPARE MSG TO ALL RMs, SETS A TIMEOUT : max WAIT TIME FOR RM's ANSWER

1.2 **RMs**: WAIT FOR THE PREPARE MESSAGE, WHEN RECEIVED IT:

- ✓ RELIABLE STATE : WRITE READY RECORD IN LOG + SEND READY MESSAGE TO TM
- * NOT RELIABLE STATE: SEND NOT READY TO TM + TERMINATE PROTOCOL + LOCAL ROLLBACK
- ✗ RM CRASH : NO ANSWER SENT

1.3 **TM** COLLECTS ALL INCOMING MESSAGE FROM RMs:

- ✓ ALL RMs READY : COMMIT GLOBAL DECISION RECORD WRITTEN ON LOG
- ✗ IF ∃ NOT READY FROM RMs : ABORT GLOBAL DECISION WRITTEN ON LOG

2.

2.1 **TM** : SENDS GLOBAL DECISION TO ALL RMs + SETS A TIMEOUT FOR RMs ANSWER

2.2 **RMs** : WAIT FOR GLOBAL DECISION, WHEN IT RECEIVES IT :

→ COMMIT / ABORT RECORD WRITTEN ON LOG + DB UPDATED + ACK TO TM

2.3 **TM**: COLLECTS ACKs FROM ALL RMs

- ✓ ALL ACK RECEIVED: COMPLETE RECORD ON LOG
- ✗ TIMEOUT EXPIRES AND ∃ ACK MISSING:

NEW TIMEOUT SET + GLOBAL DECISION SENT AGAIN TO RM WITH NO ANSWER

- UNCERTAINTY WINDOW : RMs LOCKED DURING THIS TIME
- **FAILURE OF RM**: READY LIST (LIST OF TRANS. ID's IN READY STATE) + RECOVERY REQUEST TO TM
- **FAILURE OF TM**:

- PREPARE IS LAST RECORD IN TM LOG: GLOBAL ABORT DECISION
- GLOBAL DECISION IS LAST RECORD IN TM: REPEAT PHASE 2

- **MAP REDUCE:**
  - 2 FUNCTIONS:

$$\begin{cases} \text{MAP}: & \forall doc \mapsto \langle K : [v_1, \dots, v_n] \rangle \\ \text{REDUCE}: & \forall \langle K : [v_1, \dots, v_n] \rangle \mapsto \langle K : \oplus([v_1, \dots, v_n]) \rangle \end{cases}$$

- **MAP**: ∀ RECORD IN DOCUMENT ⟶ PRODUCES < KEY : VALUES LIST >
- **REDUCE**: ∀ KEY ⟶ VALUES LIST REDUCED TO A SINGLE VALUE, BY AN AGGREGATION FUNCTION

- **MAP**:

```
function(doc) {
    emit(key_1, value_1); // key_1 = f_k1(doc); value_1 = f_v1(doc)
    emit(key_2, value_2); // key_2 = f_k2(doc); value_2 = f_v2(doc)
}
```

es.

```
Function(doc){
    emit(doc.exam, doc.mark);
}
```

Key | Value

Result:

doc.id	Key	Value
6	Bioinformatics	30
2	Computer architectures	24
3	Computer architectures	27
1	Database	29
4	Database	26
8	Database	25
5	Software engineering	21
7	Software engineering	18

*Ordered list of students, with mark and CFU for each exam

```
Function(doc) {
    key = doc.student
    value = [doc.mark, doc.CFU]
    emit(key, value);
}
```

Result:

doc.id	Key	Value
1	S123456	[29, 8]
2	S123456	[24, 10]
5	S123456	[21, 8]
6	S123456	[30, 6]
3	S654321	[27, 10]
4	S654321	[26, 8]
7	S654321	[18, 8]
8	5987654	[25, 8]

- **REDUCE**:

es.

*Map - List of exams and corresponding mark

```
Function(doc){
    emit(doc.exam, doc.mark);
}
```

*Reduce - Compute the average mark for each exam

```
Function(key, values){
    S = sum(values);
    N = len(values);
    AVG = S/N;
    return AVG;
}
```

The reduce function receives:
- key=Bioinformatics, values=[30]
- key=Database, values=[29,26,25]

Map

doc.id	Key	Value
6	Bioinformatics	30
2	Computer architectures	24
3	Computer architectures	27
1	Database	29
4	Database	26
8	Database	25
5	Software engineering	21
7	Software engineering	18

Reduce

Key	Value
Bioinformatics	30
Computer architectures	25.5
Database	26.67
Software engineering	19.5

*Average mark the for each exam (**group level=1**)

DB

doc.id	Key	Value
6	Bioinformatics, 2015-16	30
2	Computer architectures, 2015-16	24
3	Computer architectures, 2015-16	27
4	Database, 2014-2015	26
8	Database, 2014-15	25
1	Database, 2015-16	29
5	Software engineering, 2014-15	21
7	Software engineering, 2015-16	18

Map

Key	Value
Bioinformatics, 2015-16	30
Computer architectures, 2015-16	24
Computer architectures, 2015-16	27
Database, 2014-2015	26
Database, 2014-15	25
Database, 2015-16	29
Software engineering, 2014-15	21
Software engineering, 2015-16	18

Reduce

Key	Value
Bioinformatics, 2015-16	30
Computer architectures, 2015-16	25.5
Database, 2014-15	25.5
Database, 2015-16	29
Software engineering, 2014-15	21
Software engineering, 2015-16	18

Rereduce

Key	Value
Bioinformatics	30
Computer architectures	25.5
Database	27.25
Software engineering	19.5

*Map - Ordered list of students, with mark and CFU for each student

```
Function(doc) {
    key = doc.student
    value = [doc.mark, doc.CFU]
    emit(key, value);
}
```

*Reduce - Average CFU-weighted mark for each student

```
Function(key, values){
    S = sum([ X*Y for X,Y in values ]);
    N = sum([ Y for X,Y in values ]);
    AVG = S/N;
    return AVG;
}
```

key = S123456, values = [[29,8], [24,10], [21,8]...], X = 29, 24, 21, ... → mark, Y = 8, 10, 8, ... → CFU

The reduce function receives:
- key=S123456, values=[[29,8], [24,10], [21,8]...]
- ...
- key=5987654, values=[[25,8]]

Map

doc.id	Key	Value
1	S123456	[29, 8]
2	S123456	[24, 10]
5	S123456	[21, 8]
6	S123456	[30, 6]
3	S654321	[27, 10]
4	S654321	[26, 8]
7	S654321	[18, 8]
8	5987654	[25, 8]

Reduce

Key	Value
S123456	25.6
S654321	23.9
5987654	25

*Compute the number of exams for each student

*Technological view of data distribution among different nodes

DB

doc.id	Key	Value
1	S123456	[29, 1]
2	S123456	[24, 1]
5	S123456	[21, 1]
6	S123456	[30, 1]
3	S654321	[27, 1]
4	S654321	[26, 1]
7	S654321	[18, 1]
8	5987654	[25, 1]

Map

Key	Value
S123456	3
S123456	1
S654321	3
S654321	1
5987654	1

Reduce

Key	Value
S123456	4
S654321	3
5987654	1

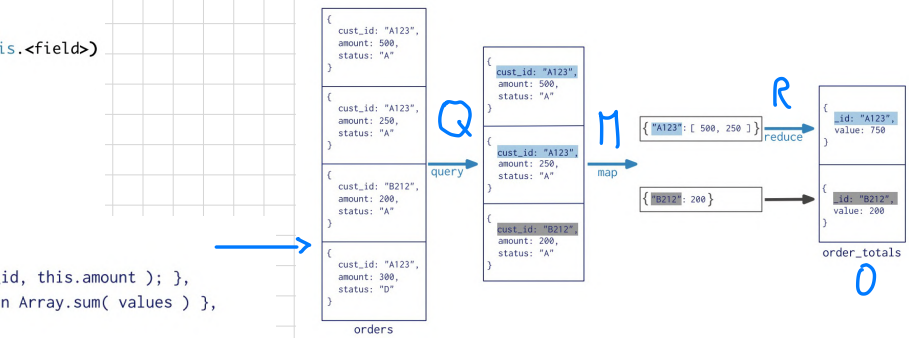
Rereduce

(MONGODB MAP REDUCE + HADOOP)

```
db.<nameDB>.mapReduce(
    // map
    function(){
      emit(this.<field>, this.<field>)
    }
    // reduce
    function(key, value){
      return f(value)
    }
    // query:
    query: {<query>}
    // output
    output: <output name>
    )
```

Collection

```
db.orders.mapReduce(
    M map      → function() { emit( this.cust_id, this.amount ); },
    R reduce   → function(key, values) { return Array.sum( values ) },
    Q query    → {
                   query: { status: "A" },
    O output   →   out: "order_totals"
                 }
)
```



- DESIGN RECIPE : BANKING ACCOUNT

- DB MUST BE SERIOUS, NO ERRORS ALLOWED
- THEY WON'T LOSE MONEY OR CREATE NEW ONE

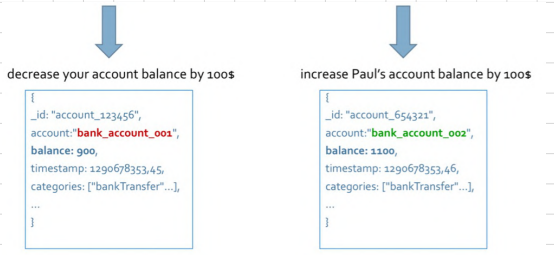
-> MUST BE IN BALANCE ALL THE TIME

EX.

A WANTS TO GIVE 100 $ TO B

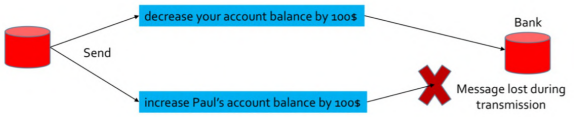
-> NEED TO
- + 100 $ ON B BANK ACCOUNT
- - 100 $ ON A BANK ACCOUNT

L> 2 SEPARATE UPDATES



- WHAT IF FAILURES HAPPENS ?

EX. MESSAGE LOST DURING TRANSMISSION



—> NoSQL DB CAN'T GARANTEE BANK BALANCE

- SOLUTION : ABOUT A METHOD WHICH DOESN'T USE 2 PHASE COMMIT

SOL -> STORE THE TRANSACTION, NOT ACCOUNT BALANCE

- HOW TO READ ACCOUNT BALANCE :

```
Map
    function(transaction){
     emit(transaction.from, transaction.amount*-1);
     emit(transaction.to, transaction.amount);
    }
Reduce
    function(key, values){
     return sum(values);
    }
Result
  {rows: [ {key: "bank_account_001", value: 900} ]
  {rows: [ {key: "bank_account_002", value: 1100} ]
```

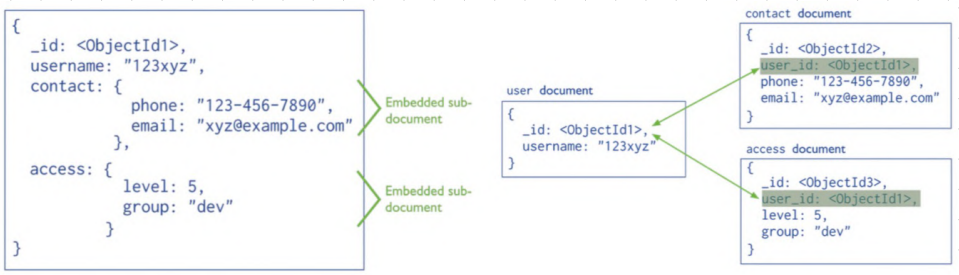
The reduce function receives:
- key= bank_account_001, values=[1000, -100]
- ...
- key= bank_account_002, values=[1000, 100]
- ...

```
id:     transaction001
from:   "bank_account_001",
to:     "bank_account_002",
qty:    100,
when:   1290678353.45,
...
```

DESIGN PATTERNS 1 :

- NoSQL DB DOESN'T REQUIRE A SCHEMA ∀ DOCUMENTS
  → IT IS POSSIBLE TO SET RULES FOR DOCUMENT VALIDATION

- EMBEDDED vs REFERENCE :



- ATOMICITY OF WRITE OPERATIONS :
  - ATOMICITY IN WRITE OPERATION <=> UPDATES (1 OR MORE) ON A SINGLE DOCUMENT
  - WHEN A SINGLE WRITE OPERATION UPDATES N DOCUMENTS
    → OPERATION IS NOT ATOMIC AS A WHOLE

  → IN MONGO DB ∃ SPECIFIC MULTI-DOCUMENT TRANSACTIONS WHICH PROVIDE ATOMICITY

- SCHEMA VALIDATION :
  - MONGO DB PERFORMS VALIDATION DURING UPDATES AND INSERT
    → DOESN'T VALIDATE PRE-EXISTING DOCUMENT UNLESS UPDATED
    - VALIDATOR : SPECIFY RULES OR EXPRESSION FOR THE COLLECTION
    - VALIDATIONLEVEL : HOW STRICTLY MONGO DB APPLIES VALIDATION RULES TO ∃ DOCUMENTS
      - STRICT : ∀ DOCUMENT → CHANGES APPLIED (DEFAULT)
      - MODERATE : CHANGES APPLIED <=> EXISTING DOCUMENTS SATISFY VALIDATION CRITERIA
    - VALIDATIONACTION : DETERMINES IF MONGO DB SHOULD OR NOT RAISE ERRORS AND REJECT DOCUMENTS THAT VIOLATE VALIDATION RULES OR GIVE ANY WARNING

```
db.createCollection( <name>,
        {validator: <document>,
        validationLevel: <string>,
        validationAction: <string>,
        })
```

- IT IS POSSIBLE TO USE THE JSON SCHEMA VALIDATOR AS VALIDATOR:

```
db.createCollection("students",
        { validator: {
                $jsonSchema: {
                        bsonType: "object",
                        required: [ "name", "year" ],
                        properties: {
                                name: {
                                        bsonType: "string",
                                        description: "must be a string and is required"
                                },
                                year: {
                                        bsonType: "int",
                                        minimum: 2000,
                                        maximum: 2099,
                                        description: "must be an integer in [2000, 2099] and is required»
                                }
                        }
                }
        }
})
```

- QUERY EXPRESSION SCHEMA VALIDATOR:

```
db.createCollection( "contacts",
        { validator: {
                $or: [
                        { phone: { $type: "string" } },
                        { email: { $regex: /@mongodb\.com$/ } },
                        { status: { $in: [ "Unknown", "Incomplete" ] } }
                ]
        }
} )
```

- DESIGNING FACTORS:
  - ATOMICITY: EMBEDDED DATA MODEL VS MULTI-DOCUMENT TRANSACTION
  - SHARDING: ~> DB PARTITIONING
    SHARD KEY MUST BE SELECTED PROPERLY -> PERFORMANCE, QUERY ISOLATION, WRITE SCALABILITY
  - INDEXES: ∀ INDEX -> AT LEAST 8 KB
    SOMETIMES ADDING INDEXES CAN HAVE NEGATIVE IMPACT ON PERFORMANCES
    COLLECTION WITH HIGH R/W OPERATIONS BENEFITS FROM INDEXES
    IF ACTIVE, INDEXES CONSUMES DISK SPACE AND MEMORY
  - DATA LIFECYCLE MANAGEMENT: SELECTION OF TIME TO LIVE OF DOCUMENTS

· BUILDING WITH PATTERNS :

1. APPROXIMATION :

ex. UPDATE +1 IN COUNTER

SOME UPDATES DOESN'T NEED TO BE PRECISE, BUT IT CAN BE IMPORTANT TO KNOW A ROUGH VALUE: ex. # PEOPLE WHO VISITED A WEB PAGE

PEOPLE IN A CITY

-> IF UPDATES CAN BE AVOIDED -> ↑ PERFORMANCE

ex. POPULATION COUNTER, MOVIE WEBSITE COUNTER

2. ATTRIBUTE :

IF DOCUMENTS SHARE SIMILAR FIELDS

-> CREATE COLLECTION

ex. PRODUCT CATALOGUE

```
{
    title: "Star Wars",
    director: "George Lucas",
    ...
    release_US: ISODate("1977-05-20T01:00:00+01:00"),
    release_France: ISODate("1977-10-19T01:00:00+01:00"),
    release_Italy: ISODate("1977-10-20T01:00:00+01:00"),
    release_UK: ISODate("1977-12-27T01:00:00+01:00"),
    ...
}
```

↳ ex. RELEASE DATE

```
{
    title: "Star Wars",
    director: "George Lucas",
    ...
    releases: [
        {
        location: "USA",
        date: ISODate("1977-05-20T01:00:00+01:00")
        },
        {
        location: "France",
        date: ISODate("1977-10-19T01:00:00+01:00")
        },
        {
        location: "Italy",
        date: ISODate("1977-10-20T01:00:00+01:00")
        },
        {
        location: "UK",
        date: ISODate("1977-12-27T01:00:00+01:00")
        },
        ...
    ],
    ...
}
```

3. BUCKET:

STREAM OF DATA CAN BE "BUCKET" TOGETHER AND ADDITIONAL INFO CAN BE COMPUTED

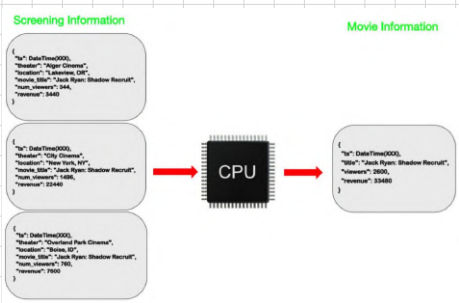
ex. IoT, TIME SERIES

```
{
    sensor_id: 12345,
    timestamp: ISODate("2019-01-31T10:00:00.000Z"),
    temperature: 40
}

{
    sensor_id: 12345,
    timestamp: ISODate("2019-01-31T10:01:00.000Z"),
    temperature: 40
}

{
    sensor_id: 12345,
    timestamp: ISODate("2019-01-31T10:02:00.000Z"),
    temperature: 41
}
```

```
{
    sensor_id: 12345,
    start_date: ISODate("2019-01-31T10:00:00.000Z"),
    end_date: ISODate("2019-01-31T10:59:59.000Z"),
    measurements: [
        {
        timestamp: ISODate("2019-01-31T10:00:00.000Z"),
        temperature: 40
        },
        {
        timestamp: ISODate("2019-01-31T10:01:00.000Z"),
        temperature: 40
        },
        ...
        {
        timestamp: ISODate("2019-01-31T10:42:00.000Z"),
        temperature: 42
        }
    ],
    transaction_count: 42,
    sum_temperature: 2413
}
```

4. COMPUTED :

IT CAN BE USEFUL TO STORE PRECOMPUTED DATA IN DB IN ORDER TO COMPUTE VALUES EACH TIME



ex. REVENUE OR VIEWER, TIME SERIES DATA, PRODUCT CATALOGS

5. DOCUMENT VERSIONING :

IT IS USEFUL TO STORE PREVIOUS VERSIONS OF OBJ

- "REVISION" / "VERSION" ATTRIBUTE CAN BE USEFUL IN SOME DOCUMENTS FOR SOME APPLICATIONS

EX. FINANCIAL INDUSTRIES, HEALTHCARE INDUSTRIES

6. EXTENDED REFERENCE :

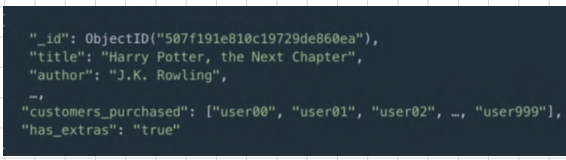
USEFUL WHEN LOT OF JOINS NEEDED

- INSTEAD OF EMBED FULL DOCUMENT IN ANOTHER, ONLY EMBED MOST FREQUENT ACCESSED

PART 2

7. OUTLIERS :

MANAGE SITUATION WHERE 3 SOME DOCUMENTS WITH SPECIAL CHARACTERISTICS



EX. BESTSELLER BOOK WITH TOUCHED BUYERS -> TOO BIG IN DIMENSIONS

- SOL : FLAG DOCUMENT AS OUTLIER AND MOVE EXTRA INFO IN ANOTHER DOCUMENT

EX. SOCIAL MEDIA RELATIONSHIP (INFLUENCER), BOOK SALES, MOVIE REVIEWS

8. PRE-ALLOCATION :

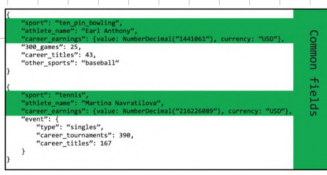
FIND DATA STRUCTURES WHICH BETTER ALLOW ACCESS TO DATA

EX. 2-DIMENSIONAL STRUCTURES, RESERVATION SYSTEM

9. POLYMORPHIC :

GROUP OBJECT THAT ARE SIMILAR TOGETHER, BASED ON THE QUERIES TO RUN

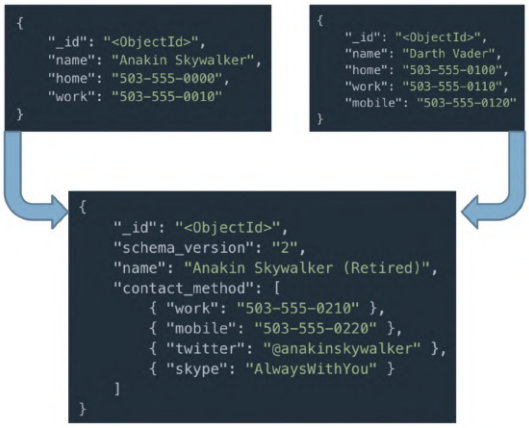
EX. SINGLE VIEW APPLICATION, CROSS-COMPANY OR CROSS-UNIT USE CASE, WIDE PRODUCT CATALOG



10. SCHEMA VERSIONING :

HAVING AN "id" FOR EACH SCHEMA VERSION

A DOCUMENT CAN HELP THE DB HOW TO HANDLE THE DOCUMENT AND MAKE CHANGES EASILY MANAGEBLE



EX. CUSTOMER PROFILE

11. SUBSET :

IF LARGE DOCUMENTS OR LARGE COLLECTION SIZE

-> USEFUL TO CREATE SUBSET TO REDUCE RAM WORKLOAD

- A COLLECTION CAN BE SPLITTED IN 2 :
  MOST FREQUENT DOCUMENTS / LEAST FREQUENT DOCUMENTS

EX. REVIEW OF A PRODUCT (COLLECTION SPLIT : ONLY MOST RECENT REVIEWS)

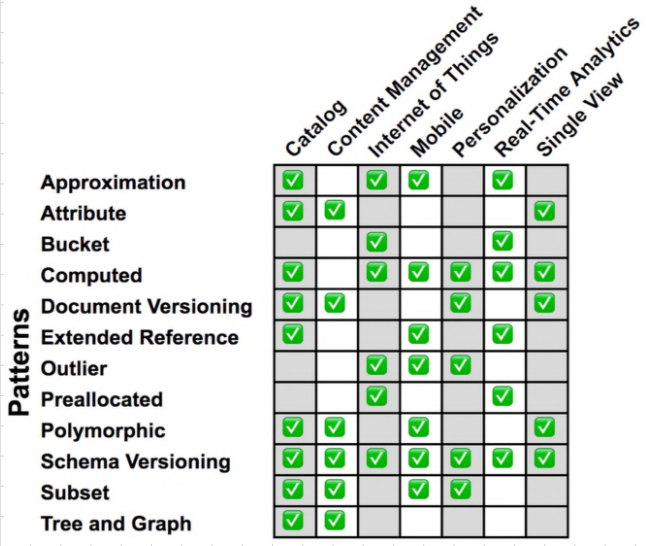
12. TREE :

TREE HIERARCHY STRUCTURE MANAGEMENT

- INSTEAD OF STORING ∀ NODE -> PARENT / CHILDREN
  ↳ STORE ∀ DOCUMENT ITS PATH

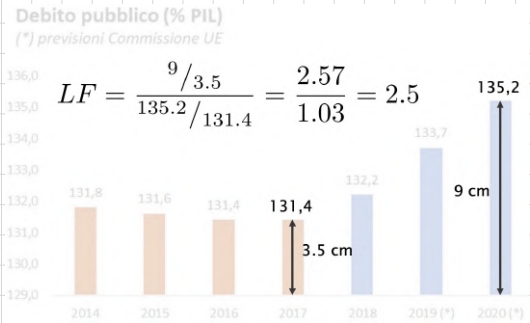
```
{
  _id: <ObjectId1>,
  name: "Samsung 860 EVO 1 TB Internal",
  part_no:  "MZ-76E1T0B ",
  price: {
     value: NumberDecimal("169.99 "),
     currency: "USD"
  },
  parent_category: "Solid State Drives",
  ancestor_categories: [
     "Solid State Drives",
     "Hard Drives",
     "Storage",
     "Computers",
     "Electronics"
  ]
}
```

- SUMMARY :

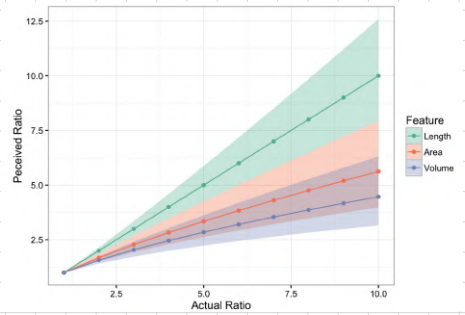


- **DATA VISUALIZATION (UTILITY):**
  - LENGTH ARE EASIER TO COMPARE THAN AREAS (PIE CHART ↦ BAR CHART)
    - ↳ (AND VOLUME WORSE THAN AREAS)
  - **PROPORTIONALITY:**
    - **LIE FACTOR:**

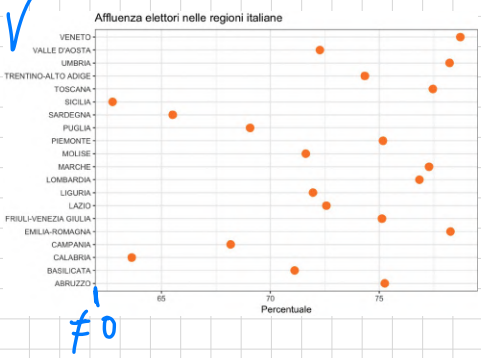
$$LF = \frac{\text{FRACTION IN GRAPH}}{\text{FRACTION IN DATA}} \mapsto 1$$



    - LF: OK | UNDERSTATING | OVERSTATING
      - ↳ by (LP) → 0
    - **DATA-INK RATIO:** $\frac{\text{DATA INK}}{\text{TOTAL INK IN GRAPHIC}}$ ⟶ MAXIMIZE IT
  - **CLARITY:**
    - FOR TRENDS → USE LINES, NOT BARS
    - **WEBER'S LAW:** $d_p(x) = K_p \times$ (JUST NOTICEABLE DIFFERENCE; CONST.; ↳ DIMENSION)
      - ↳ $x < y$, $d_p(x) < d_p(y)$
    - **STEVENS'S LAW:** $p(x) = c \cdot x^{\beta}$ (↳ DIMENSION PERCEIVED; → e.g. 1D, 2D, 3D)



- **GRAPH CONSTRUCTION:**
  - WHEN RANKING -> HORIZONTAL BAR
  - PIE CHART <=> MAX 4 CATEGORIES, PROPORTIONS USE DISTINGUISHABLE
  - HUE ~> COLORS IN RAINBOW
    - ~~HUE~~ NO ORDERING / SATURATION, LUMINOSITY HUE ORDERING
  - DIRECT LABELS > LEGEND
  - DOT / BAR PLOT: DOT CAN BE USED FOR COMPARISON WITHOUT 0 SCALE BASE



✓

≠ 0

- **DATA QUALITY:**
  - CWA vs OWA: CLOSED AND NON CLOSED WORLDS