Armazéns de Dados

Departamento de Engenharia Informática (DEI/ISEP)
Paulo Oliveira
pjo@isep.ipp.pt

Adaptado do Original de:
Fátima Rodrigues (DEI/ISEP)

1

Data Warehouse Optimization

2

Bibliography

- The Data Warehouse Lifecycle Toolkit : Expert Methods for Designing, Developing, and Deploying Data Warehouses
  Ralph Kimball, Laura Reeves, Margy Ross, Warren Thornthwaite
  Wiley, 1998

  Chapters 14, 15

- Modern Database Management
  J.Hoffer, M.Prescott, H. Topi
  Prentice Hall, 2008

  Chapter 6

Indexing

Indexes

- In almost all DWs, the **size of dimension tables is insignificant compared with the size of fact tables**
- **Second biggest thing in a DW** is the **size of indexes of the fact tables**
- Indexing mechanisms are used to **speed up access** to desired data
- Index file consists of records (called index entries) of the form:

search-key	pointer

- **Search key** is **an attribute** or **set of attributes** used to **look up records** in a file and **pointer points to the record** in the data table
- One of the most powerful capabilities of indexed file organization is the ability to create **multiple indexes**

5

5

Index Types

- **B-Tree index**
  - Adequate for **high cardinality attributes**
  - May be built **on multiple columns**
  - Is the **default index type** for most databases, created on the primary key of a table
- **Bitmap index**
  - Adequate for **not high cardinality attributes** (*e.g.*: marital status)
  - Are a **major advance in indexing** that benefit DW applications
  - Are used both with **dimension tables and fact tables**, where the constraint on the table results in a **not high cardinality**
- **Others**
  - **Hash indexes** - array of *n* buckets or slots, each one containing a pointer to a row.
  - Some DBMSs use **additional index structures or optimizations** strategies **adequate to the n-way join** problem, inherent to a star query (*e.g.*: Red Brick: star indexes).

6

6

Bitmap Index

- Bitmap is simply an **array of bits**
- Bitmap index on an attribute has a **bit for each value of the attribute**
  - Bitmap has as many bits as distinct values
  - In a Bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

Id_Client	Gender	City	Income Level
145023	M	Brooklin	L1
145025	F	Jonestown	L2
154265	F	Perryridge	L4
265453	M	Brooklin	L1
645654	F	Perryridge	L3

B-Tree index   Bitmap index

Bitmap Index Structure

- Bitmap index for *Color* and *Type*

Cars			
ID	Type	Color	..other..
1DGS902	Sedan	White	...
1HUE039	Sedan	Silver	...
2UUE384	Coupe	Red	...
2ZUD923	Coupe	White	...
3ABD038	Sedan	Silver	...
3KES734	Coupe	White	...
3IEK299	Sedan	Red	...
3JSU823	Sedan	Silver	...
3LOP929	Coupe	Silver	...
3LMN347	Coupe	Red	...
3SDF293	Sedan	White	...

Color Bit Map Index		
Silver	Red	White
0	0	1
1	0	0
0	1	0
0	0	1
1	0	0
0	0	1
0	1	0
1	0	0
1	0	0
0	1	0
0	0	1

Type Bit Map Index	
Sedan	Coupe
1	0
1	0
0	1
0	1
1	0
0	1
1	0
1	0
0	1
0	1
1	0

- Bitmap index **often requires less storage space** than a conventional B-tree index
- For an attribute with **many distinct values**, can **exceed the storage space** of a conventional B-tree index

Using Bitmap Indexes

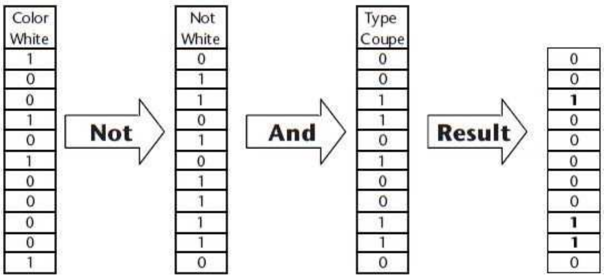
- Bitmap indexes **are useful for queries on multiple attributes** but not particularly useful for single attribute queries
- Queries are answered using **bitmap operations**
  - Intersection (AND)
  - Union (OR)
  - Negation (NOT)
- Each operation takes a bitmap vector and **applies the operation** to get the result bitmap vector
- Bit manipulation and searching is so fast, that the speed of **query processing with a bitmap index can be 10 times faster** than with a conventional B-tree index
- Databases support: DB2; Informix; Ingres; Oracle; PostgreSQL

9

9

Queries on Bitmap Indexes

- Example of a query: find all cars that are not white and which are coupes



Cars			
ID	Type	Color	..other..
1DGS902	Sedan	White	...
1HUE039	Sedan	Silver	...
2UUE384	**Coupe**	**Red**	**...**
2ZUD923	Coupe	White	...
3ABD038	Sedan	Silver	...
3KES734	Coupe	White	...
3IEK299	Sedan	Red	...
3JSU823	Sedan	Silver	...
3LOP929	**Coupe**	**Silver**	**...**
3LMN347	**Coupe**	**Red**	**...**
3SDF293	Sedan	White	...

- Database is able to resolve the query using the **bitmap vectors** and **Boolean operations**
- It does not need to **touch the data** until it has **isolated the rows that answer the query**

10

10

Managing Indexes

- Indexes are performance enhancers at query time, but **performance killers at insert and update time**
- **Most efficient procedure to follow when loading a fact or a dimension table:**
  - Segregate inserts from updates
  - Drop any indexes not required to support updates
  - Perform the updates
  - Drop all remaining indexes
  - Perform the inserts
  - Rebuild the indexes

11

11

Indexing Dimension Tables

- Dimension tables have a **single column primary key** – must have **one unique index on that key**
- Small dimension tables seldom benefit from additional indexing
- Large dimension tables (*e.g.*: customer, product)
  - **Single-column bitmap or B-tree indexes** on **dimension attributes** that are most commonly used for:
    - **applying filters** (only makes sense in ROLAP)
    - **grouping** (only makes sense in ROLAP)
    - **used in a join condition**

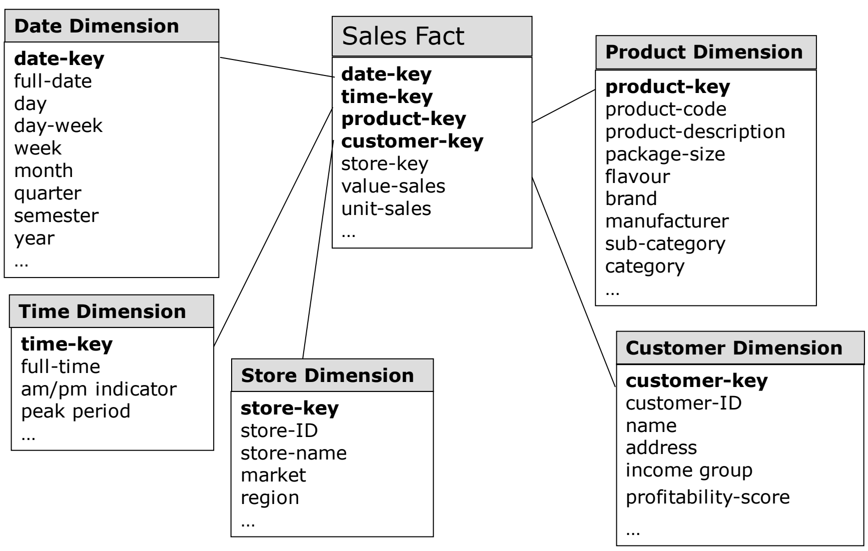
12

12

Indexing Fact Tables

- Fact table index must be a **B-Tree index** on the **primary key**
- Primary key, and the primary key index, must have **date-key in the first position** in the primary key index
  - Incremental loads are keyed by date
  - Most DW queries (in ROLAP) are constrained by date
- Create a **single-column index on each fact table key** and let the optimizer combine those indexes as appropriate to answer the queries
  - Only makes sense in ROLAP
- If many queries constrain fact column values (amount, quantity) they must also be included in indexes – **non-key fact table indexes** are **single-column indexes**
  - Only makes sense in ROLAP

How to Index a Store Sales Star Schema?



Initial Index Plan for the Store Sales Schema

Date Dimension

Index Name	Index Type	Unique	Columns	Justification
calendar-pkey	B-Tree	Y	date-key	Primary key index
calendar-fulldate	B-Tree	Y	full-date	Used in joins to load fact table(s)
calendar-month	Bitmap	N	month	Used in group by clause
calendar-year	Bitmap	N	year	Dimension filtering and group by

Time Dimension

Index Name	Index Type	Unique	Columns	Justification
time-pkey	B-Tree	Y	time-key	Primary key index
time-fulltime	B-Tree	Y	full-time	Used in joins to load fact table(s)

Initial Index Plan for the Store Sales Schema

Product Dimension

Index Name	Index Type	Unique	Columns	Justification
prod-pkey	B-Tree	Y	product-key	Primary key index
prod-code	B-Tree	N	product-code	Used in joins to load fact table(s)
prod-description	B-Tree	N	product-description	Dimension filtering and group by
prod-brand	Bitmap	N	brand-name	Dimension filtering and group by
prod-subcategory	Bitmap	N	Subcategory-name	Dimension filtering and group by
prod-category	Bitmap	N	Category-name	Dimension filtering and group by
prod-package-size	Bitmap	N	Package-size	Dimension filtering and group by

Initial Index Plan for the Store Sales Schema

Customer Dimension

Index Name	Index Type	Unique	Columns	Justification
cust-pkey	B-Tree	Y	customer-key	Primary key index
cust-ID	BTree	N	Customer-ID	Used in joins to load fact table(s)
cust-state	Bitmap	N	State_code	Dimension filtering and group by
cust-income	Bitmap	N	Income-group	Dimension filtering and group by
cust-profitability	Bitmap	N	Profitability-score	Dimension filtering and group by

Initial Index Plan for the Store Sales Schema

Store Dimension

Index Name	Index Type	Unique	Columns	Justification
store-pkey	B-Tree	Y	store-key	Primary key index
store-market	Bitmap	N	market	Dimension filtering and group by
store-region	Bitmap	N	region	Dimension filtering and group by
store-country	Bitmap	N	country	Dimension filtering and group by

Initial Index Plan for the Store Sales Schema

Sales Fact

Index Name	Index Type	Unique	Columns	Justification
sales-pkey	B-Tree	Y	Date-key Time-key Product-key Customer-key	Primary key index
sales-date	B-Tree	N	Date-key	Used in most star-join user queries
sales-time	B-Tree	N	Time-key	Used in most star-join user queries
sales-product	B-Tree	N	Product-key	Used in most star-join user queries
sales-customer	B-Tree	N	Customer-key	Used in most star-join user queries
sales-store	Bitmap	N	Store-key	Used in most star-join user queries

Partitions

Partitions

- Segments tables in **small tables** for administrative purposes and to **improve performance**
  - SQL instructions manipulate the partitions instead of all the table
- Partitions are particularly useful
  - When they are defined in **different disks**
  - To manage **big fact tables**, **large dimensions**, and **their indexes**
- **Index Partitions**
  - **Global Index** – indexes all rows, regardless of the partition
  - **Local Index** – indexes only rows in a specific partition
- DBMSs support **different combinations** of table and index partitions
  - Partitioned table with index partitioned
  - Partitioned table with index not partitioned
  - Not partitioned table with index partitioned

21

21

Horizontal Partitions

- Breaks a table by **placing different rows into different physical files** based on a condition
- Makes sense when **different categories of rows** of a table **are processed separately**
- **Best way to horizontally partition tables is by date**, with data segmented by year, semester, or quarter into separate storage partitions
  - ➥Date is often a qualifier in queries so, the **needed partitions are quickly found**
- Each file created from horizontal partitioning **has the same structure**

22

22

Vertical Partitions

- **Distributes the columns of a table into separate physical records**, repeating the primary key in each of the records
- **Schema is different** from partition to partition
- **Less used** than horizontal partitions
- Reasons to use:
  - **Performance** – Updates and queries perform better because the database is able to buffer more rows at a time
  - **Change history** – some attributes change more frequently
  - **Large text** – placing large text columns in their own tables

23

23

Methods of Horizontal Partitions

- **Range partitioning**
  - Partitions are created according to a **range of values for an attribute or a set of attributes**
  - Assumes that the **values** of the attributes used in the partition respect an **order**
- Best results occur when:
  - Size of partitions are **uniform**
  - Access to data is distributed through a **small number of partitions**

```
CREATE TABLE sales_range
(sales_id NUMBER(5),
sales_name VARCHAR(30),
sales_amount NUMBER(10),
sales_date DATE)
PARTITION BY RANGE (sales_date)
(PARTITION sales_2019 VALUES LESS THAN(TO_DATE('01/01/2020','DD/MM/YYYY')) tablespace ts1,
 PARTITION sales_2020 VALUES LESS THAN(TO_DATE('01/01/2021','DD/MM/YYYY')) tablespace ts2,
 PARTITION sales_2021 VALUES LESS THAN(TO_DATE('01/01/2022','DD/MM/YYYY')) tablespace ts3,
 PARTITION sales_2022 VALUES LESS THAN(TO_DATE('01/01/2023','DD/MM/YYYY')) tablespace ts4);
```

24

24

Methods of Horizontal Partitions

- **List partitioning**
  - Partitions are created according to a **list of values of an attribute** explicitly specified
  - Useful when **data is not ordered nor has a relation**
  - **Values** of the attribute used in the partition **must be previously known**

```
CREATE TABLE sales_list
(sales_id NUMBER(5),
 sales_name VARCHAR(30),
 sales_region VARCHAR(20),
 sales_amount NUMBER(10),
 sales_date DATE)
PARTITION BY LIST(sales_region)
( PARTITION sales_north VALUES('Porto', 'Braga', 'Vila Real') tablespace ts1,
  PARTITION sales_central VALUES ('Lisboa', 'Santarém', 'Setúbal') tablespace
  ts2,
  PARTITION sales_south VALUES('Beja', 'Faro') tablespace ts3,
  PARTITION sales_other VALUES(DEFAULT) tablespace ts4);
```


Partitions in Data Warehouses

- Table partitions can **dramatically improve query performance on large fact tables**
  - Performing a query that **requires a year of data from a partitioning fact table** that has ten years of data, can go **directly to the partition that contains the data** without scanning all the table
- Common practice: **partitioning the fact table by date** (e.g., year, semester, quarter)
- **Partitions are maintained** by the DB administrator or by the DW administrator

Aggregates

27

Aggregates

- **Summarization of a set of measures**, stored into fact tables with the purpose of **accelerating queries**
- Always associated with **one or more dimensions** that are **aggregated or not**
  - Example: **Sales** by ***product category***, by ***store*** , by ***date***

    Fact   aggregate dimension   dimensions

- Can have a very significant **effect on performance**
  - Speeding queries by a factor that goes from 100 to 1000
  - No other means exist to provide such performance gain

28

28

Regular vs Aggregate Dimension

Regular Date Dimension

Date Dimension
DateKey
Fulldate
Year
Semester
Quarter
Month
MonthName
WeekOfYear
WeekOfMonth
DayNumberOfYear
DayNumberOfMonth
DayNumberOfWeek
DayOfWeek
Weekend
…

Aggregate Month Dimension

Month Dimension
MonthKey
Year
Semester
Quarter
Month
MonthName

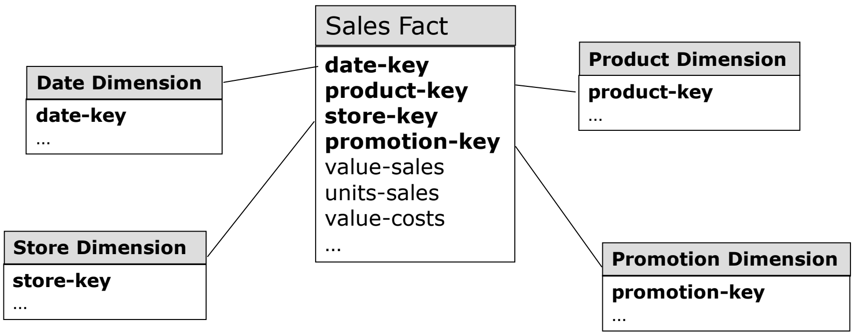
Aggregates Goals/Requirements

- Aggregates in a large DW must do **more than just improving performance**
  - Provide **huge performance gains** for as many categories of user queries as possible
  - Add only **reasonable amount of extra data storage** to the DW
    - Increase the overall disk space by a **factor of two or less**
  - Be **completely transparent to end-users** and application designers, *i.e.*, no end-user application should reference the aggregates
  - Directly **benefit all users of the DW**, regardless of which query/analysis tool they use
  - Impact the **data loading tasks** as little as possible

Aggregates Advantages/Disadvantages

- **Advantages**
  - Improve performance of the DW
  - Transparent to end-users and applications
  - Shared by many users
- **Disadvantages**
  - Only speed up the answers for the questions previously known, i.e., previously calculated and stored
  - Need constant attention by the DW administrator to:
    - Build new aggregates adequate to the more frequent questions made by the users
    - Eliminate aggregates that are not useful
  - Spend disk space

Aggregates: Grocery Store



- Aggregates:
  - Totals by category (product dimension)
  - Totals by region (store dimension)
  - Totals by month (date dimension)
- How many **aggregate fact tables** are necessary?

Final Scheme: Grocery Store

- **Star base**: Fact table + 4 dimensions
- **3 aggregate dimensions**:
  - Category (Product dimension)
  - Region (Store dimension)
  - Month (Date dimension)
- **Aggregate fact tables**:
  1. Totals by category, store, year
  2. Totals by category, store, month
  3. Totals by category, region, year
  4. Totals by category, region, month
  5. Totals by region, product, year
  6. Totals by region, product, month
  7. Totals by store, product, year
  8. …

These tables are not visible to end-users

33

33

Deciding What to Aggregate

- Two different areas need to be considered when selecting which aggregates to build
  - **Common business users' requests**
    - Major geographic groupings (regions, districts, …)
    - Major product groupings (category, subcategory, …)
    - Regular reporting time periods groupings (months, quarters, …)
    - …
    - Combinations of these attributes
  - **Statistical distribution of data**

34

34

Common Business Users' Requests

- Reviews should be performed to determine **which attributes are commonly used for grouping**, considering:
  - Each attribute individually
  - Combinations of attributes
    - Within a dimension
    - Among dimensions
- Not all attributes or combinations of attributes **are used together**

35

35

Statistical Distribution of Data

- **Number of attribute values** that are **candidates for aggregation**
  - 1.000.000 products exist in the product dimension
    - Aggregates at next level 500.000 – would not provide a significant improvement
    - Level that aggregates to 75.000 would be a strong pre-stored aggregate
  - Date dimension (5 years)

Date dimension (5 years)		Product dimension	
Day	**1826**	**SKU**	**2023**
Month	**60**	**Product**	**723**
Quarter	**20**	**Brand**	**44**
Year	**5**	**Category**	**15**

    - Month aggregate alone cuts data to 1/30 of detail size
    - Brand aggregate cuts data to about 1/50 of the detail size

36

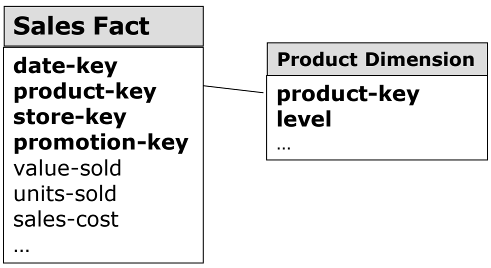
36

Building Aggregates

- Aggregates can be built for a period of time by:
  - **Adding the current atomic load records to existing accumulating buckets** in the staging area
  - ***Group by* operation over the transactional data** in the operational system or staging area
  - ***Group by* operation over the DW data** that has already been loaded

Techniques to Store Aggregates

1. Aggregate facts and aggregate dimensions are stored in **new tables, separated from the base atomic data**
2. Aggregate facts are stored in the atomic fact table, and **level attributes** are stored in the dimensions
   - *Level* attribute show the aggregation level of each row



- Original rows are filled with *level* = 'Base'
- Aggregates for *Category* are filled with the *level* = '*Category*'

Comparing the Two Methods

- Number of records created **is the same** in both methods
- **Separated Tables**
  - Tables that correspond to the aggregates are not visible to end-users
  - Aggregates in separated tables can be easily created, deleted, loaded and indexed
- **Level Attributes**
  - Can conduct a **double-count additive facts totals** – all the queries must restrict the level attribute – if not, all the values are included/added
  - **Wasting disk storage** – adding the aggregates in the base fact table implies to increase the attribute width for all the records
  - **Dimension tables are more complicated** – for the records corresponding to the aggregates, many attributes are filled with '*not applicable'* or with *null*

39